

MULTI: Scripting



Green Hills Software, Inc.

**30 West Sola Street
Santa Barbara, California 93101
USA**

Tel: 805-965-6044

Fax: 805-965-6343

www.ghs.com

DISCLAIMER

GREEN HILLS SOFTWARE, INC. MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. Further, Green Hills Software, Inc. reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Green Hills Software, Inc. to notify any person of such revision or changes.

Copyright © 1983-2009 by Green Hills Software, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission from Green Hills Software, Inc.

Green Hills, the Green Hills logo, CodeBalance, GMART, GSTART, INTEGRITY, MULTI, and Slingshot are registered trademarks of Green Hills Software, Inc. AdaMULTI, Built with INTEGRITY, EventAnalyzer, G-Cover, GHnet, GHnetLite, Green Hills Probe, Integrate, ISIM, u-velOSity, PathAnalyzer, Quick Start, ResourceAnalyzer, Safety Critical Products, SuperTrace Probe, TimeMachine, TotalDeveloper, DoubleCheck, and velOSity are trademarks of Green Hills Software, Inc.

All other company, product, or service names mentioned in this book may be trademarks or service marks of their respective owners.

PubID: 10417
December 12, 2009

Contents

	Preface	xix
	About This Book	xx
	The MULTI 5 Document Set	xxi
	Conventions Used in the MULTI Document Set	xxii
1	Introduction	1
	The MULTI Integrated Development Environment	2
	MULTI Launcher	2
	Editing Tools	2
	Building Tools	2
	Debugging Tools	3
	Administrative Tools	3
	The MULTI Launcher	4
	MULTI Workspaces	5
	Accessing Online Help	6
	Full Online Manuals	6
	Context-Sensitive Help	6
	Command and Configuration Option Help	6
	The Help Viewer Window	7
	Navigating Manuals in the Help Viewer	8
Part I	MULTI Scripting	
2	Using MULTI Scripts	15
	Script Overview	16
	Creating a MULTI Script	17
	Using Macros	17
	Example Scripts	18
	Checking the Syntax of Your Script	23
	Running a Script	24

Contents

Part II MULTI-Python Scripting

3	Introduction to the MULTI-Python Integration	29
	Python Installation	30
	The Default Python Installation	30
	Using a Customized Python Installation	31
	MULTI-Python Compatibility	32
	Overview of MULTI-Python Classes	32
	GHS_IdeObject Attributes	32
	MULTI-Python Service Classes	33
	GHS_Window Attributes	34
	MULTI-Python Window Classes	35
	MULTI-Python Debugger Object Classes	36
	MULTI-Python Utility Classes	36
	MULTI-Python Miscellaneous Classes	37
	MULTI-Python Class Hierarchy	37
	MULTI-Python Utility Functions	43
	MULTI-Python Variables	44
	Pre-Set Variables	44
	Reserved Variable Names	44
	Extending the MULTI-Python Environment	46
	MULTI-Python Interfaces	47
	Interface Comparison	49
	Py Pane Commands and Keyboard Shortcuts	50
	The mpythonrun Utility Program	53
	mpythonrun Command Line Options	53
	Running Python Statements and Scripts	55
	Starting Socket Servers	56
	Socket Server Commands	57
	Creating a Graphical Interface	58
	Troubleshooting	59
4	MULTI-Python Tutorials	61
	Manipulating Windows	62
	Manipulating the Editor	75
	Manipulating the Debugger	77

Contents

	Using Tcl/Tk to Create a Graphical Interface	85
Part III	MULTI-Python API Reference	
5	General Notes on Using Functions	89
6	MULTI-Python Utility Function Prototypes	91
	Utility Function Prototypes	92
	GHS_ExecFile()	92
	GHS_PrintObject()	93
	GHS_RunShellCommands()	93
	GHS_System()	94
7	Basic Functions	97
	GHS_IdeObject Functions	98
	CleanCmdExecVariables()	98
	IsAlive()	98
8	Window Functions	99
	GHS_Window Basic Functions	101
	GetCwd()	101
	GetInfo()	101
	GetPid()	101
	IsSameWindow()	102
	IsWindowAlive()	102
	RunCommands()	102
	GHS_Window Configuration Functions	103
	ClearDefaultConfigFile()	103
	LoadConfigFile()	103
	SaveConfig()	104
	ShowConfigWindow()	105
	GHS_Window Directory Functions	105
	GetIntegrityDistributionDir()	105
	SetIntegrityDistributionDir()	105
	GetUvelocityDistributionDir()	106
	SetUvelocityDistributionDir()	106
	GetLatestDir()	106
	SetLatestDir()	107
	GHS_Window Interactive Functions	108

Contents

Beep()	108
ChooseDir()	108
ChooseFile()	109
ChooseFromList()	110
ChooseWindowFromGui()	110
ChooseYesNo()	111
GetInput()	111
ShowMessage()	112
Wait()	112
GHS_Window Menu Functions	113
DumpMenu()	113
DumpMenuBar()	114
GetCommandToDumpMenu()	114
GetCommandToDumpMenuBar()	115
GetCommandToSelectMenu()	115
GetCommandToSelectMenuPath()	116
GetCommandToSelectSubMenu()	116
GetCommandToSelectSubSubMenu()	117
IsMenuItemActive()	118
IsMenuItemTicked()	118
IsSubMenuItemActive()	119
IsSubMenuItemTicked()	119
SelectMenu()	120
SelectSubMenu()	120
SelectSubSubMenu()	121
WaitForMenuItem()	122
GHS_Window Modal Dialog Functions	123
GetCommandToRegisterModalDialogCommands()	123
RegisterModalDialogCommands()	124
RegisterModalDialogToChangePullDownValue()	124
RegisterModalDialogToClickButton()	125
RegisterModalDialogToDoubleClickMslCell()	125
RegisterModalDialogToDumpWidget()	126
RegisterModalDialogToDumpWindow()	127
RegisterModalDialogToSelectMslCell()	127
RegisterModalDialogToSelectMslCellByValue()	128
RegisterModalDialogToSelectPullDownMenu()	129
RegisterModalDialogToShowWidgets()	129
RegisterModalDialogToSortMsl()	130
RemoveRegisteredModalDialogCommands()	130

Contents

ShowRegisteredModalDialogCommands()	131
GHS_Window Record Functions	132
RecordGuiOperations()	132
GHS_Window Window Attribute and Manipulation Functions	133
CloseWindow()	133
GetDimension()	133
GetName()	133
GetPosition()	134
IconifyWindow()	134
IsIconified()	135
MoveWindow()	135
RenameWindow()	135
ResizeWindow()	136
RestoreWindow()	137
ShowAttributes()	137
9 Widget Functions	139
GHS_MslTree Attributes and Functions	141
DumpTree()	142
GetChildrenNumber()	142
IsExpandable()	143
IsExpanded()	143
IsTopTree()	143
SearchByColumnValue()	143
SearchByName()	144
SearchChildByColumnValue()	145
SearchChildByName()	145
SearchRow()	146
GHS_Window Basic Widget Functions	146
DumpAll()	146
DumpWidget()	147
GetCommandToDumpWidget()	147
GetCommandToDumpWindow()	148
GetCommandToShowWidgets()	149
ShowWidgets()	149
GHS_Window Button Widget Functions	151
DumpButton()	151
GetCommandToClickButton()	152
GetCommandToDumpButton()	152

Contents

IsButtonDimmed()	153
IsButtonDown()	153
SelectButton()	153
WaitButtonInStatus()	154
GHS_Window Edit and Terminal Widget Functions	155
GetEditTextLines()	155
GetEditTextString()	155
GHS_Window MScrollList Widget Functions	156
ChangeMslTree()	156
ChangeWholeMslTree()	157
DoubleClickMslCell()	157
DoubleClickMslCellByValue()	158
DumpMslHighlight()	158
DumpMslSelection()	159
DumpMslValue()	159
ExtendMslSelection()	160
GetCommandToChangeMslTree()	161
GetCommandToDoubleClickMslCell()	161
GetCommandToDoubleClickMslCellByValue()	162
GetCommandToDumpMsl()	163
GetCommandToDumpMslHighlight()	163
GetCommandToDumpMslSelection()	164
GetCommandToExtendMslSelection()	164
GetCommandToSelectMslCell()	165
GetCommandToSelectMslCellByValue()	166
GetCommandToSortMsl()	166
GetMslRowNumber()	167
GetMslTree()	167
SelectMslCell()	168
SelectMslCellByValue()	168
SortMslByColumn()	169
GHS_Window PullDown Widget Functions	170
ChangePullDownValue()	170
DumpPullDownMenu()	170
DumpPullDownValue()	171
GetCommandToChangePullDownValue()	171
GetCommandToDumpPullDownMenu()	172
GetCommandToDumpPullDownValue()	172
GetCommandToSelectPullDownMenu()	173
GetPullDownMenu()	173

Contents

GetPullDownValue()	174
SelectPullDownValue()	174
GHS_Window Tab Widget Functions	174
DumpTabContents()	175
DumpTabSelection()	175
DumpTabValue()	176
GetCommandToDumpTab()	176
GetCommandToDumpTabSelection()	177
GetCommandToDumpTabValue()	177
GetCommandToSelectTab()	178
GetTabNames()	179
GetTabSelection()	179
SelectTab()	179
GHS_Window Text Widget Functions	180
GetCommandToDumpText()	180
GetTextValue()	181
GHS_Window TextCell Widget Functions	181
DumpTextCellValue()	181
GetTextCellValue()	182
IsTextCellReadOnly()	182
GHS_Window TextField Widget Functions	183
ChangeTextFieldValue()	183
DumpTextFieldValue()	183
GetCommandToChangeTextFieldValue()	184
GetCommandToDumpTextField()	185
GetCommandToReturnOnTextField()	185
GetTextFieldValue()	186
IsTextFieldReadOnly()	186
ReturnOnTextField()	186
10 Window Tracking Functions	189
GHS_WindowRegister Basic Functions	190
__init__()	190
GHS_WindowRegister Check Functions	191
CheckWindow()	191
CheckWindowObject()	192
IsWindowInList()	192
GHS_WindowRegister Get Window Functions	193
GetCheckoutBrowserWindow()	193

Contents

GetConnectionOrganizerWindow()	194
GetDebuggerWindow()	194
GetDialogByName()	194
GetDiffViewerWindow()	195
GetEditorWindow()	195
GetEventAnalyzerWindow()	195
GetHelpViewerWindow()	196
GetLauncherWindow()	196
GetOsaExplorerWindow()	196
GetProjectManagerWindow()	197
GetPythonGuiWindow()	197
GetResourceAnalyzerWindow()	197
GetTaskManagerWindow()	198
GetTerminalWindow()	198
GetTraceWindow()	198
GetWindowByIndex()	199
GetWindowByName()	199
GetWindowList()	199
ShowWindowList()	200
GHS_WindowRegister Interactive Functions	201
Beep()	201
ChooseDir()	201
ChooseFile()	202
ChooseFromList()	203
ChooseWindowFromGui()	203
ChooseYesNo()	204
GetInput()	205
ShowMessage()	205
GHS_WindowRegister Window Manipulation Functions	206
CloseAllWindows()	206
IconifyAllWindows()	206
RestoreAllWindows()	206
GHS_WindowRegister Wait Functions	207
WaitForWindow()	207
WaitForWindowFromClass()	208
WaitForWindowGoAway()	208
WaitForWindowObjectGoAway()	209
11 Connection Functions	211
GHS_DebuggerApi Target Connection Functions	213

Contents

ConnectToRtserv()	213
ConnectToRtserv2()	214
ConnectToTarget()	214
Disconnect()	215
IsConnected()	215
GHS_DebuggerApi Window Display Functions	216
ShowConnectionOrganizerWindow()	216
GHS_DebugServer Functions	216
__init__()	216
Disconnect()	217
GetComponent()	217
LoadProgram()	217
RunCommands()	218
ShowTaskManagerWindow()	218
GHS_Terminal Functions	219
__init__()	219
MakeConnection()	219
GHS_TerminalWindow Functions	220
ChangeBaudRate()	220
Connect()	220
Disconnect()	220
SendBreak()	221
12 Debug Functions	223
GHS_Debugger Functions	225
__init__()	225
RunCommands()	226
GHS_DebuggerApi Debug Flag Functions	226
ChangeBreakpointInheritance()	226
ChangeDebugChildren()	227
ChangeDebugOnTaskCreation()	227
ChangeHaltOnAttach()	228
ChangeInheritProcessBits()	229
ChangeRunOnDetach()	229
ChangeStopAfterExec()	230
ChangeStopAfterFork()	230
ChangeStopOnTaskCreation()	231
CheckBreakpointInheritance()	232
CheckDebugChildren()	232

Contents

CheckDebugOnTaskCreation()	232
CheckHaltOnAttach()	233
CheckInheritProcessBits()	233
CheckRunOnDetach()	233
CheckStopAfterExec()	234
CheckStopAfterFork()	234
CheckStopOnTaskCreation()	234
GHS_DebuggerApi Host Information Functions	235
GetHostOsName()	235
GetMultiVersion()	235
GHS_DebuggerApi Memory Access Functions	236
ReadIndirectValue()	236
ReadIntegerFromMemory()	236
ReadStringFromMemory()	237
WriteIntegerToMemory()	237
WriteStringToMemory()	238
GHS_DebuggerApi Run-Control Attributes and Functions	238
DebugProgram()	239
GetPc()	240
GetProgram()	240
GetStatus()	240
GetTargetPid()	240
Halt()	240
IsHalted()	241
IsRunning()	241
IsStarted()	241
Kill()	242
Next()	242
Resume()	243
Step()	243
WaitToStop()	244
GHS_DebuggerApi Symbol Functions	244
CheckSymbol()	244
GetSymbolAddress()	245
GetSymbolSize()	245
GHS_DebuggerApi Target Information Functions	245
BigEndianTarget()	245
GetCpuFamily()	246

Contents

GetCpuMinor()	246
GetTargetCoProcessor()	246
GetTargetCpuFamilyName()	247
GetTargetId()	247
GetTargetOsMinorType()	247
GetTargetOsName()	247
GetTargetOsType()	248
GetTargetSeries()	248
IsFreezeMode()	248
IsNativeDebugging()	248
IsRunMode()	249
GHS_DebuggerApi Window Display Functions	249
ShowOsaExplorerWindow()	249
ShowTaskManagerWindow()	250
ShowTraceWindow()	250
GHS_DebuggerWindow Basic Functions	250
RunCommands()	250
GHS_DebuggerWindow Breakpoint Functions	251
RemoveBreakpoint()	251
SetBreakpoint()	252
SetGroupBreakpoint()	252
ShowBreakpoints()	253
ShowBreakpointWindow()	253
GHS_DebuggerWindow Print Functions	254
DumpToFile()	254
PrintFile()	254
PrintWindow()	255
GHS_MemorySpaces Attributes and Functions	255
__init__()	256
GHS_OsTypes Attributes and Functions	256
__init__()	257
GHS_TargetIds Functions	257
__init__()	257
GHS_Task Basic Functions	258
__init__()	258
RunCommands()	258
RunCommandsViaDebugServer()	259
GHS_Task Run-Control Functions	260

Contents

Attach()	260
Detach()	260
Halt()	261
Next()	261
Resume()	262
Step()	262
GHS_TraceWindow Functions	263
FlushTraceBuffer()	263
JumpToTrigger()	263
StartTracing()	263
StopTracing()	264
13 Editor Functions	265
GHS_Editor Functions	266
__init__()	266
EditFile()	267
GotoLine()	267
GHS_EditorWindow Edit Functions	268
AddString()	268
Copy()	268
Cut()	269
GetTextLines()	269
GetTextString()	269
Paste()	270
Redo()	270
Undo()	270
GHS_EditorWindow File Functions	271
CloseCurrentFile()	271
GotoNextFile()	271
GotoPrevFile()	271
OpenFile()	272
SaveAsFile()	272
SaveIntoFile()	273
GHS_EditorWindow Selection and Cursor Functions	273
FlashCursor()	273
GetSelection()	273
GetSelectedString()	274
MoveCursor()	274
SelectAll()	275
SelectRange()	275

Contents

	GHS_EditorWindow Version Control Functions	276
	Checkin()	276
	Checkout()	276
	PlaceUnderVC()	276
14	EventAnalyzer Functions	277
	GHS_EventAnalyzer Functions	278
	__init__()	278
	CloseFile()	279
	GetFileList()	279
	OpenFile()	279
	ScrollToPosition()	280
	GHS_EventAnalyzerWindow File Functions	281
	CloseFile()	281
	OpenFile()	281
	GHS_EventAnalyzerWindow View and Selection Functions	282
	GotoFirstView()	282
	GotoLastView()	282
	GotoNextView()	283
	GotoPrevView()	283
	SelectRange()	284
	ToggleFlatView()	284
	ViewRange()	285
	ZoomIn()	285
	ZoomOut()	285
	ZoomToSelection()	285
	GHS_EventAnalyzerWindow Miscellaneous Functions	286
	AutoTimeUnit()	286
	ChangeTimeUnit()	286
	NewWindow()	287
	SaveMevConfiguration()	287
	ShowLegend()	287
15	Launcher Functions	289
	GHS_Action Attributes and Functions	291
	DumpTree()	291
	GHS_ActionSequence Attributes and Functions	292
	DumpTree()	292
	Search()	292

Contents

GHS_Variable Attributes and Functions	293
DumpTree()	293
GHS_Workspace Attributes and Functions	294
DumpTree()	295
SearchAction()	295
SearchActionSequence()	296
SearchVariable()	296
GHS_Launcher Functions	297
__init__()	297
GHS_LauncherWindow Action Execution Functions	298
GetRunningActions()	298
RunAction()	298
RunWorkspaceAction()	299
WaitForActionsToFinish()	300
GHS_LauncherWindow Action Manipulation Functions	301
AddAction()	301
DeleteAction()	302
GHS_LauncherWindow Workspace Manipulation Functions	303
CreateWorkspace()	303
DeleteWorkspace()	303
GetWorkspaceInformation()	304
GetWorkspaces()	305
LoadWorkspaceFile()	305
SaveWorkspaceIntoFile()	306
SelectWorkspace()	306
GHS_LauncherWindow Variable Functions	307
AddVariable()	307
ChangeVariable()	308
DeleteVariable()	308
16 Project Manager Functions	311
GHS_ProjectManager Functions	312
__init__()	312
GetTopProjectFiles()	313
OpenProject()	313
GHS_ProjectManagerWindow Build Functions	314
BuildAllProjects()	314
BuildFile()	314

Contents

BuildProjects()	315
BuildSelectedProjects()	316
GetStatus()	316
HaltBuild()	316
WaitForBuildingFinish()	317
GHS_ProjectManagerWindow Edit Functions	317
CopySelected()	317
CutSelected()	318
DeleteSelected()	318
PasteAfterSelected()	318
GHS_ProjectManagerWindow File Functions	318
CloseProject()	318
NewWindow()	319
OpenProject()	319
RevertFromFile()	320
SaveChanges()	320
GHS_ProjectManagerWindow Navigation Functions	321
FindFile()	321
NextFile()	321
GHS_ProjectManagerWindow Debug and Edit Functions	322
DebugSelectedProjects()	322
EditSelectedProjects()	322
GHS_ProjectManagerWindow Tree Expansion/Contraction Functions	323
ContractAll()	323
ContractSelected()	323
ExpandAll()	324
ExpandSelected()	324
PrintAll()	324
PrintView()	325
GHS_ProjectManagerWindow Selection Functions	325
DoubleClickTreeRow()	325
SelectAll()	326
SelectProject()	326
SelectTreeRow()	327
17 Version Control Functions	329
GHS_CoBrowse Functions	330

Contents

__init__()	330
OpenCheckoutBrowserWindow()	331
GHS_DiffView Functions	331
__init__()	331
DiffFiles()	332
OpenChooseWindow()	332
GHS_DiffViewWindow Basic Functions	333
OpenDiff()	333
OpenNewDiff()	333
ShowCurrentDiff()	333
GHS_DiffViewWindow Display Functions	334
ToggleCaseSensitive()	334
ToggleCharDiff()	334
ToggleIgnoreAllWhiteSpace()	334
ToggleIgnoreCWhiteSpace()	335
ToggleIgnoreWhiteSpaceAmount()	335
ToggleLineupColumns()	335
ToggleNumber()	336
ToggleWordDiff()	336
18 Miscellaneous Functions	337
GHS_AbortExecFile and GHS_AbortExecFileWithStack Functions	339
__init__()	339
GHS_AbortExecOnSignal Functions	339
__init__()	339
GHS_Exception Functions	340
__init__()	340
GHS_WindowClassNames Attributes and Functions	340
__init__()	341
 Part IV Appendix	
A Third-Party License and Copyright Information	345
PSF License Agreement for Python 2.3	346
Tcl/Tk License Terms	347
BLT Copyright Information	348

Contents

Index

351

Contents

Preface

This Preface Contains:

- About This Book
- The MULTI 5 Document Set
- Conventions Used in the MULTI Document Set

This section discusses the purpose of the manual, typographical conventions used, and the MULTI documentation set.

About This Book

This book contains information about creating MULTI scripts and about the MULTI-Python integration. It is divided into the following parts:

- *Part I: MULTI Scripting* describes how to use MULTI's built-in scripting language. See page 13.
- *Part II: MULTI-Python Scripting* describes how to use MULTI's integration with Python. It also contains tutorials that demonstrate how to use the MULTI-Python integration to control MULTI IDE components. See page 27.
- *Part III: MULTI-Python API Reference* describes all the Python functions provided in the MULTI-Python API. See page 87.
- *Part IV: Appendix* contains license and copyright information for third-party tools shipped with MULTI. See page 343.



Note New or updated information may have become available while this book was in production. For additional material that was not available at press time, or for revisions that may have become necessary since this book was printed, please check your MULTI installation directory for release notes, **README** files, and other supplementary documentation.

The MULTI 5 Document Set

The primary documentation for using MULTI is provided in the following books:

- *MULTI: Getting Started* — Describes how to install MULTI, and takes you through creating, building, and debugging an example project.
- *MULTI: Licensing* — Describes how to obtain, install, and administer Green Hills licenses. This book is intended for system administrators.
- *MULTI: Building Applications* — Describes how to use the MULTI Project Manager and compiler drivers and the tools that compile, assemble, and link your code. Also describes the Green Hills implementation of supported high-level languages.
- *MULTI: Configuring Connections* — Describes how to set up your target debugging interface for use with MULTI and how to configure connections to your target.
- *MULTI: Debugging* — Describes how to use the MULTI Debugger and its associated tools.
- *MULTI: Debugging Command Reference* — Describes how to use Debugger commands and provides a comprehensive reference of Debugger commands.
- *MULTI: Editing Files and Configuring the IDE* — Describes how to use the MULTI Editor and MULTI Launcher, how to use a version control system with MULTI, and how to configure the MULTI IDE.
- *MULTI: Scripting* — Describes how to create MULTI scripts and contains information about the MULTI-Python integration.

For a comprehensive list of the books provided with your MULTI installation, see the **toc_target.pdf** file located in the **manuals** subdirectory of your installation.

Most books are available in the following formats:

- A printed book (select books are not available in print).
- Online help, accessible from most MULTI windows via the **Help** → **Manuals** menu.
- An electronic PDF, available in the **manuals** subdirectory of your installation.

Conventions Used in the MULTI Document Set

All Green Hills documentation assumes that you have a working knowledge of your host operating system and its conventions, including its command line and graphical user interface (GUI) modes. For example, you should know how to use basic commands, how to open, save, and close files, and how to use a mouse and standard menus.

Green Hills documentation uses a variety of notational conventions to present information and describe procedures. These conventions are described below.

Convention	Meaning	Example
bold type	Indicates a: • Filename or pathname • Command • Option • Window title • Menu name or menu choice • Field name • Button name	• C:\MyProjects • setup command • -G option • The Breakpoints window • The File menu • Working Directory: • The Browse button
<i>italic type</i>	Indicates: • Replaceable text • A new term • A book title	• -o <i>filename</i> • A task may be called a <i>process</i> or a <i>thread</i> • <i>MULTI: Debugging</i>

Convention	Meaning	Example
monospace type	Indicates: • Text you should enter as presented • A word or words used in a command or example • Source code • Input/output • A function	• Type <code>help</code> <code>command_name</code> • The wait [-global] command blocks command processing, where -global blocks command processing for all MULTI processes. • <code>int a = 3;</code> • <code>> print Test</code> <code>Test</code> • <code>GHS_System()</code>
ellipsis (...) (in command line instructions)	Indicates that the preceding argument or option can be repeated zero or more times.	debugbutton [name]...
greater than sign (>)	Represents a prompt. Your actual prompt may be a different symbol or string. The > prompt helps to distinguish input from output in examples of screen displays.	<code>> print Test</code> <code>Test</code>
pipe () (in command line instructions)	Indicates that one (and only one) of the parameters or options separated by the pipe or pipes should be specified.	call <i>proc</i> <i>expr</i>
square brackets ([]) (in command line instructions)	Indicate optional arguments, commands, options, and so on. You can either include or omit the enclosed elements. The square brackets should not appear in your actual command.	<code>.macro</code> <i>name</i> [<i>list</i>]

The following command description demonstrates the use of some of these typographical conventions.

gxyz [-option]... *filename*

The formatting of this command indicates that:

- The command **gxyz** should be entered as shown.
- The option *-option* should either be replaced with one or more appropriate options or be omitted.
- The word *filename* should be replaced with the actual filename of an appropriate file.

The square brackets and the ellipsis should not appear in the actual command you enter.

Introduction

Chapter 1

This Chapter Contains:

- The MULTI Integrated Development Environment
- The MULTI Launcher
- Accessing Online Help

This chapter provides an overview of the MULTI Integrated Development Environment.

The MULTI Integrated Development Environment

MULTI is a complete Integrated Development Environment (IDE) designed especially for embedded systems engineers to assist them in compiling, optimizing, debugging, and editing embedded applications.

The MULTI IDE includes the following tools for each part of the software development process. Many of the MULTI tools can be launched from within the IDE or as separate stand-alone programs. Parenthetical text indicates corresponding executable files, which are located in your MULTI installation.

MULTI Launcher

MULTI Launcher (mstart) — The gateway to the MULTI IDE, which allows you to quickly launch any of the primary MULTI tools, access open windows, and manage MULTI workspaces.

Editing Tools

- **MULTI Editor (me)** — A graphical editor for modifying text files.
- **Checkout Browser (mcobrowse)** — A graphical viewer for files managed under a version control system.
- **Diff Viewer (diffview)** — A graphical viewer that displays differences between two text files.

Building Tools

- **MULTI Project Manager (mprojmgr)** — A graphical interface for managing and building projects.
- **Integrate (integrate)** — A graphical utility for configuring tasks, connections, and kernel objects across multiple AddressSpaces when using the INTEGRITY RTOS.

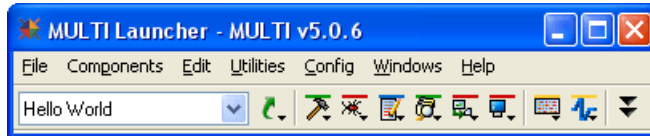
Debugging Tools

- **MULTI Debugger (multi)** — A graphical source-level debugger.
- **Instruction Set Simulator** — A tool that simulates your embedded processor on your development host.
- **EventAnalyzer (mevgui)** — A graphical viewer for monitoring the complex real-time interactions of an embedded RTOS such as INTEGRITY or u-velOSity.
- **ResourceAnalyzer (wperf)** — A graphical viewer for monitoring the CPU and memory usage of an embedded system running the INTEGRITY RTOS.
- **Serial Terminal (mterminal)** — A serial terminal emulator for connecting to serial ports on embedded devices.
- **DoubleCheck** — A static source code analysis tool for locating programming problems that lead to security and reliability failures in software.
- **MULTI TimeMachine Tool Suite** — A wide variety of trace analysis tools that dramatically extend MULTI's trace and debugging capabilities.

Administrative Tools

- **Bug Report (gbugrpt)** — A utility for providing system configuration and tool version information to the Green Hills support staff.
- **Green Hills Probe Administrator (gpadmin)** — A graphical interface for configuring and managing Green Hills Probes and SuperTrace Probes.
- **Graphical Utilities (wgutils)** — A collection of utilities for analyzing and performing various operations on object files, libraries, and executables produced with the Green Hills toolchain.
- **MULTI License Administrator (mlmadmin)** — A graphical utility for managing Green Hills tools licenses.

The MULTI Launcher



The MULTI Launcher (**mstart**) provides a convenient way to launch frequently used tools, to create new or access recently used files and projects, and to manage MULTI workspaces. All of the main MULTI components can be accessed using the following buttons:

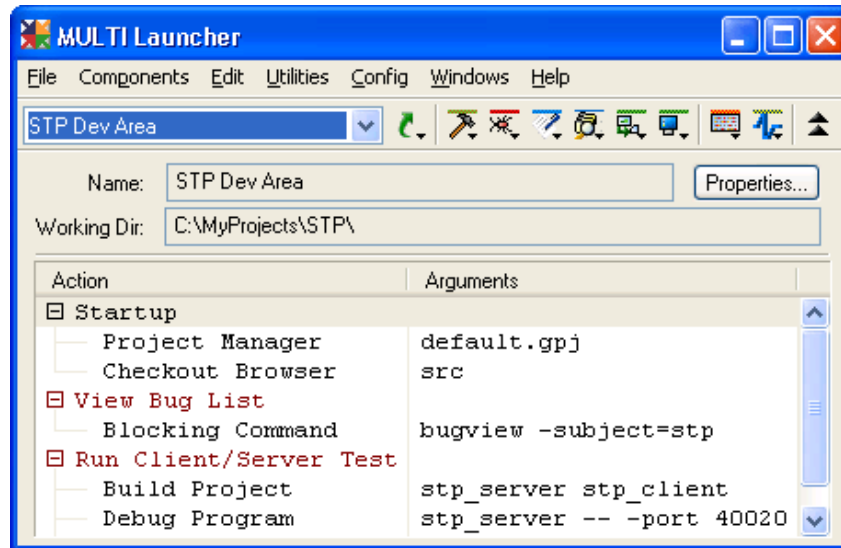
-  — Runs a shortcut or an action sequence in the current workspace. Also allows you to create a new workspace or create or edit a shortcut.
-  — Opens the Project Manager on a recent or new project.
-  — Opens the Debugger on a recent or new executable.
-  — Opens the Editor on a recent or new file.
-  — Opens the Checkout Browser on a recent or new checkout.
-  — Establishes a target connection or opens the Connection Chooser or Connection Organizer.
-  — Opens a Serial Terminal using a recent or new connection.
-  — Opens the EventAnalyzer (licensed separately).
-  — Opens the ResourceAnalyzer (licensed separately).
-  — Shows/hides the detail pane of the Launcher.

You can also launch the Green Hills License Administrator and (if installed) the Green Hills Probe Administrator from the **Utilities** menu.

During development, you can use the MULTI Launcher as a convenient, centralized window manager. You can access any of your open MULTI windows from the **Windows** menu of the Launcher.

MULTI Workspaces

The MULTI Launcher allows you to create and use *workspaces*. A MULTI workspace is a virtual area where the tools, files, and actions required for a particular project can be organized, accessed, and executed.



A workspace is typically created for each Top Project, and includes a working directory and a group of related *actions*—for example, opening a project in the MULTI Project Manager, connecting to a target, or performing a shell command. Actions are grouped into action sequences, so that a single mouse click can perform all the actions in the specified action sequence.

For more information, see Chapter 5, “Using MULTI Workspaces and Shortcuts” in the *MULTI: Editing Files and Configuring the IDE* book.

Accessing Online Help

Online help provides useful information about your MULTI tools and can be accessed in several different ways. The following sections describe how to access online help and how to use the Help Viewer.

Full Online Manuals

You can access an indexed, hypertext version of any MULTI manual by selecting it from the list that appears in the **Help** → **Manuals** menu. The MULTI Help Viewer opens on the manual specified.

You can also access manuals from the command line. Ensure that the MULTI installation directory is in your path and enter the following command:

```
helpview pathname | -m manual_name
```

where:

- *pathname* opens the **.chm** file specified in *pathname*. You must provide a full path.
- -m *manual_name* opens the manual *manual_name*. An example manual name is "MULTI: Debugging". If the manual name contains a space as in the preceding example, you must enclose it with quotation marks.

Context-Sensitive Help

Many MULTI windows and dialog boxes are linked to specific sections of the online manuals. To view the Help Viewer page that documents an active window or dialog box, press F1.

If you are using the MULTI Editor, you can also open context-sensitive help about a button or a menu item by selecting **Help** → **Identify** and then clicking the button or selecting the menu item.

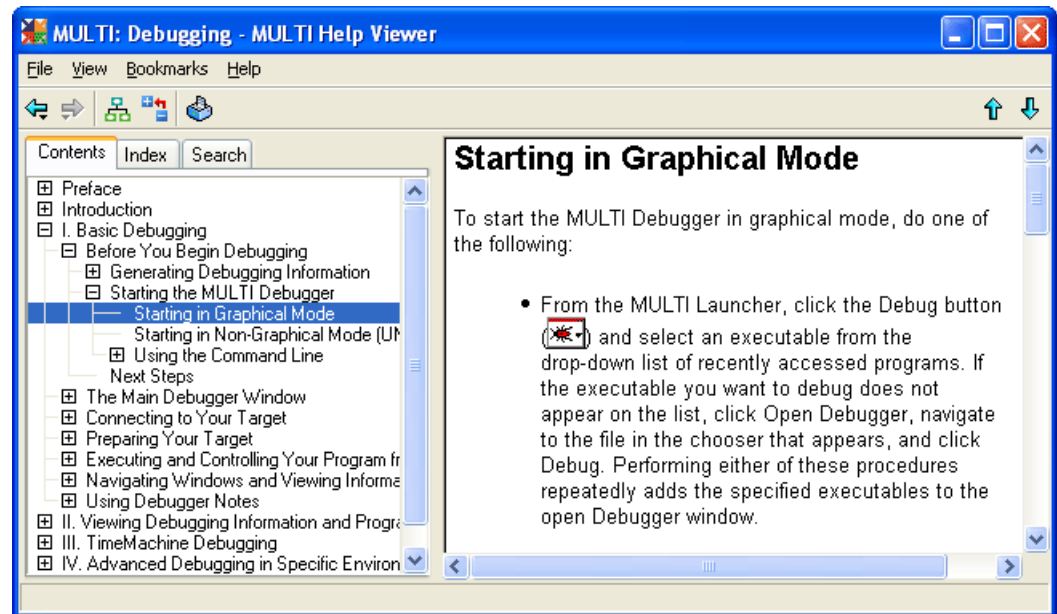
Command and Configuration Option Help

You can obtain help information about a specific MULTI Debugger command or MULTI configuration option by typing `help command_name` or `help configuration_option` in the Debugger command pane. This opens the Help

Viewer on documentation for the specified command or configuration option. You can also type `usage command_name` to print the basic syntax of a MULTI Debugger command to the command pane.

The Help Viewer Window

The following graphic displays the Help Viewer window.



The Help Viewer toolbar contains the following buttons, from left to right:

- **Back** (↶) and **Forward** (↷) — Returns to pages you have visited during the current help session. Click the button once for each page you want to revisit. Click and hold the button to select from a list of pages you have visited. These buttons function across books.
- **View All Subtrees** (📁) — Toggles highlighting of the current selection's sub-sections (if any) in the **Contents** tab, and toggles the display of help for the current selection's sub-sections (if any) in the view pane. To disable highlighting in the **Contents** tab and to display help for only the selected item, click this button again. When you open the Help Viewer, this button defaults to its last value.
- **Contract All Unselected** (📁) — Collapses hierarchies located in the **Contents** tab if the hierarchies do not contain any selections.

- **Print** () — Prints the help page displayed in the view pane. To create a help page consisting of mixed topics, hold down the Ctrl key while clicking separate topics. MULTI combines the separate topics and orders them by their location in the **Contents** tab.
- **Previous Page** () and **Next Page** () (located at far right) — Displays the previous or following section as ordered in the **Contents** tab. If the previous or next section includes sub-sections and the **View All Subtrees** button is enabled, the Help Viewer displays the sub-sections (if any) along with the section.

The MULTI Help Viewer contains two panes. The right-hand pane, or the *view pane*, displays the help page for your selection. The left-hand pane contains the following three tabs:

- **Contents** — Displays the parts, chapters, sections, and sub-sections for the manual you are viewing. Click a plus or minus icon to show or hide headings for embedded sections. Click a heading to display the associated help page in the view pane.
- **Index** — Displays index entries for the manual. To search the index entries, enter a word or words in the text box located at the top of the pane. MULTI returns results that contain the word(s) you typed. Click an index entry to display the corresponding help page in the view pane.
- **Search** — Provides an interface that allows you to search for specific words in the current manual or in all manuals.

Navigating Manuals in the Help Viewer

Navigating manuals in the MULTI Help Viewer is easy. This section describes different methods of finding information.

Viewing Previous and Next Sections

To display the previous or following section as ordered under the **Contents** tab, do one of the following:

- Click the  or  button located in the upper-right corner of the Help Viewer window.
- Select **View** → **Previous Contents Entry** or **View** → **Next Contents Entry**.

If the previous or next section includes sub-sections and the **View All Subtrees** button () is enabled, the Help Viewer displays the sub-sections (if any) along with the section.

Returning to Previously Viewed Help Pages

To return to pages you have visited during the current help session, do one of the following:

- Click the  or  button located in the left corner of the toolbar. Click the button once for each page you want to revisit. Click and hold the button to select from a complete list of pages you have visited during the current help session. These buttons function across books.
- Select **View** → **Back** or **View** → **Forward** from the menu bar. This feature functions across books.

Linking to Related Topics

The underlined links available in the view pane allow you to jump to pages related to the topic you are viewing.

If, when you click a link, a dialog box appears with the message:

You have requested information that the Help Viewer is unable to retrieve.

one of the following factors may be the cause:

- The manual that the link points to may not be included with your distribution. For a list of available manuals, look in the manuals directory (or directories) of your Green Hills Software distribution(s).
- If the link points to an INTEGRITY or u-velOSity manual, you may be using a version of the operating system that does not support links from MULTI help.
- If the link points to an INTEGRITY or u-velOSity manual, you may need to set the location of the installed OS distribution so that the Help Viewer can locate the relevant help files. For information about how to do this, see “Using MULTI with INTEGRITY or u-velOSity” in Chapter 2, “Installation” in the *MULTI: Getting Started* book.



Note If you are using MULTI with multiple target architectures (such as ARM and PowerPC), tools of the same version number should be installed into a single directory, regardless of differing architectures. Note, however, that links to target-specific books may yield unpredictable results in this situation.

Bookmarking Help Pages

You can use bookmarks to return to pages you have visited across MULTI sessions. To bookmark a page, select **Bookmarks** → **Bookmark This Page**.

You can bookmark a help page consisting of mixed topics by holding down the Ctrl key while clicking separate topics. MULTI combines the separate topics in the view pane and orders them by their location under the **Contents** tab. Bookmark the page as usual.

The **Bookmarks** menu lists all your bookmarks from across MULTI manuals and sessions. To return to a bookmarked page, do one of the following:

- Select **Bookmarks** and choose a bookmark.
- Select **Bookmarks** → **Manage Bookmarks**, click a bookmark, and click the **Show** button ().
- From any MULTI tool, select **Help** → **Bookmarks** and choose a bookmark.

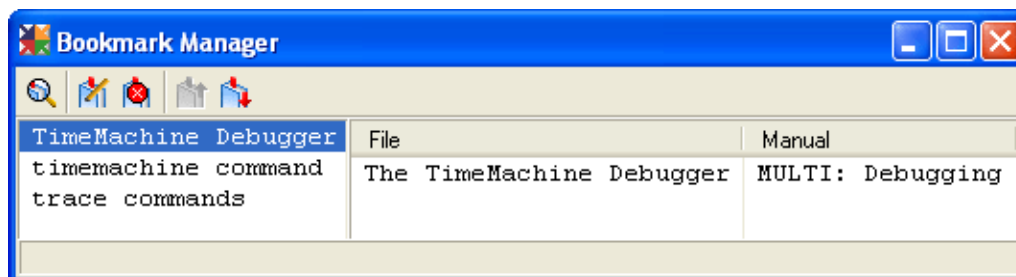
To rename, delete, or reorder existing bookmarks, launch the **Bookmark Manager** as described in the next section, “Managing Bookmarks” on page 10.

Managing Bookmarks

You can use the **Bookmark Manager** to rename and delete bookmarks, navigate to bookmarks, and modify the ordering of bookmarks in the **Bookmarks** menu.

To open the **Bookmark Manager**, do one of the following:

- In the Help Viewer, select **Bookmarks** → **Manage Bookmarks**.
- From any MULTI tool, select **Help** → **Bookmarks** → **Manage Bookmarks**.



The list on the left side of the **Bookmark Manager** displays the names of all the bookmarks that you have created. When a bookmark is selected, the file(s) and the manual marked by that bookmark are displayed in the right side of the window. The bookmark may reference multiple files (help pages) if, for example, you created the bookmark with multiple entries selected in the **Contents** tab or if you selected an index entry that pertained to multiple files.

The toolbar buttons in the **Bookmark Manager** allow you to perform the following operations on bookmarks:

- — Display the selected bookmark(s) in the Help Viewer.
- — Rename the selected bookmark.
- — Delete the selected bookmark(s).
- and — Move the selected bookmark up or down in the list. The order of the bookmarks in the **Bookmarks** menu mirrors the order of the bookmarks in the **Bookmark Manager**.



Note The and buttons are available for use on multiple bookmarks. You can select multiple bookmarks from the left side of the window by dragging to select rows or by using Shift or Ctrl to select items.

Opening Additional Manuals from the Help Viewer

To open a new manual in the current window, do one of the following:

- Select **File** → **Open Manual** and choose from the list of installed manuals.
- Select **File** → **Open Manual File** and navigate to a particular file in the file chooser that appears.

To open a new manual in a new window, select **View** → **Open Other Manuals in New Window** and then open the manual as usual. When you open the Help

Viewer, the **Open Other Manuals in New Window** toggle option defaults to its last value.

MULTI Scripting

Part I

Using MULTI Scripts

Chapter 2

This Chapter Contains:

- Script Overview
- Creating a MULTI Script
- Checking the Syntax of Your Script
- Running a Script

This chapter describes the basic conventions for writing, editing, and running scripts.

Script Overview

A script is a list of commands and expressions in a file. MULTI reads and executes this file as if you entered each command and expression individually in the Debugger command pane. Most script files end in an **.rc** extension and can be run as startup files. For more information, see “Using Script Files” in Chapter 8, “Configuring and Customizing MULTI” in the *MULTI: Editing Files and Configuring the IDE* book.

Scripts can automate common tasks and regression testing. For example, a script file can compare a program variable to a constant value and then perform some action based on the result. A script file can also execute parts of a program and then check to see whether the process ran correctly. You can use such a script to verify that your process still runs as expected even after you change the program.

Scripts can contain macros (see “Using Macros” on page 17) and flow control statements, such as `if (expr) { ... } else { ... }` or `while (expr) { ... }`. You can use these types of statements to control the execution of specific commands within a script.

You can use board setup scripts to configure your target board automatically before you download and debug code. Board setup scripts end in an **.mbs** extension. For information about using and editing board setup scripts, see “Creating Customized Setup Scripts and Linker Directives Files in Chapter 2, “Setting Up Your Target Hardware” in the *MULTI: Configuring Connections* book.

Creating a MULTI Script

You can create a script in either of the following ways:

- Enter the `> [file]` command to record commands entered in the MULTI Debugger command pane. For more information, see Chapter 16, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.
- Use a text editor.

For commands that are particularly useful for writing scripts, see Chapter 16, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book. You can also use debug server commands in your script if they are preceded by the **target** command. For more information about the **target** command, see “General Target Connection Commands” in Chapter 19, “Target Connection Command Reference” in the *MULTI: Debugging Command Reference* book. For a list of debug server commands, see Appendix A, “Green Hills Debug Server Command and Scripting Reference” in the *MULTI: Configuring Connections* book for your target processor family. Additionally, the debug server that supports your specific target may accept other commands. For additional debug server commands, see the relevant chapter in the *MULTI: Configuring Connections* book for your target processor family.

Using Macros

Use the **define** command to create macros that you can include in MULTI scripts. (See the **define** command in “Command Manipulation and Macro Commands” in Chapter 16, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.) The **define** command is similar to the C preprocessor directive `#define`. It gives you the ability to create a macro inside MULTI. You can later run that macro from the Debugger command pane or from a script. For an example of a script that references **define** macros, see Example 2 on page 19.

Example Scripts

The following examples show ways in which you can use scripts to automate common tasks.

Example 1. Connecting to a Target from MULTI

You can automate the series of repeated steps required to connect to a target. With the large number of Connection Methods possible and the diversity of specific user environments, automated connections are generally unique to each user. MULTI allows you to define exactly what you want to automate.

In the following example, suppose the target processor is a AT91RM9200, the target board is an Atmel AT91RM9200, and the probe connecting MULTI to the board is a Green Hills Probe. The program you are debugging is **foo**. The following script creates a button that connects to the target. It also includes commands that partially initialize the target. To generate a complete initialization setup script, use the **Project Wizard** to create a project for the Atmel AT91RM9200 target board.

File: **foo.rc**

```
debugbutton "My Connect" c="< connect.txt" i=connect
```

File: **connect.txt**

```
connect mpserve 192.67.44.234;  
target rst;  
target set clock 30 kHz;
```

Example 2. Regression Testing

The following regression test example consists of a program that converts Fahrenheit temperatures to Celsius.

Suppose the source file, **temp.c**, of the program you are debugging contains the code:

```
#include <stdio.h>
#define CONV (5.0/9.0)
extern int mytotal;
int celsius (int fahrenheit) {
    int rval = (int) ((fahrenheit - 32) * CONV);
    return rval;
}
void main (void) {
    int some_degrees;
    int some_celsius;
LABEL:
    some_celsius = celsius(some_degrees);
    printf("some_celsius = %i", some_celsius);
}
```

You define a script file, **temp.rc**, that *MULTI* loads every time you debug **temp**. The **temp.rc** file contains:

```
debugbutton RegTest1 c="<regtest.txt" i="letter_r"
define check_celsius(arg) {
    if (some_celsius != arg) {
        print "Failed!"
        printf ("Failed!\n actual:%d\n expected:%d\n", some_celsius, arg);
    } else {
        print "Pass";
        printf ("Pass!\n actual:%d\n expected:%d\n", some_celsius, arg);
    }
}
```

In a text file named **regtest.txt**, you define the commands that *MULTI* runs when you click the **RegTest1** button. The **regtest.txt** file contains:

```
b main##LABEL
rb
some_degrees = 45;
S
check_celsius(7);
cb
```

Now, when you start the MULTI Debugger on **temp**, MULTI runs the script **temp.rc** automatically. The script creates a button with the name **RegTest1** and the built-in icon **letter_r** with the shape of the letter “R.” The script also defines the macro `check_celsius`. When you click the **RegTest1** button, MULTI runs the commands in **regtest.txt** as if you entered them directly in the Debugger command pane. You must be connected to a target before running these commands.

The following features are demonstrated in this example:

- MULTI uses commands such as **b** as if you entered them directly in the Debugger command pane.
- The example references program variables and **define** macros. The macro defined by **temp.rc** displays the results of the regression test. For more information, see “Using Macros” on page 17.
- From your script, you can call functions that are linked into your program, such as `printf()`.
- MULTI evaluates C-like expressions such as `if() {} else {}`.

Example 3. Collecting and Printing Execution Information

The following example script single-steps 100 times and writes executed lines to the MULTI command pane, allowing you to see exactly what lines of source code and which instructions were executed. This process is similar to collecting and analyzing trace data, but is much slower and requires manual analysis. As a result, it is typically only useful if your target does not support trace.

This script works well for stepping through code that you do not have the source to (library code, for example). It does not work well if your program has timing requirements or if your program requires that interrupts be enabled.

```
$i = 100;
while ($i > 0){
    $i--;
    si;
    printline;
    wait;
}
```

You can insert other commands into the loop, as shown below:

```
$i = 100;
while ($i > 0) {
    $i--;
    si;
    wait;
    $r3;
    printline;
    wait;
}
```

In addition to single-stepping 100 times and writing executed lines to the MULTI command pane, the preceding script reads and prints register `r3` at each step, allowing you to easily track its value.

For greater flexibility, you can define a macro such as the following:

```
> define step_and_record($n){$m = $n; while ($m > 0) {
eval si; wait; printline; eval $m=$m-1 } }
> step_and_record(1000)
```

Example 4. Updating the Source Pane During Execution

Rather than printing execution information as the preceding example does, the following script updates the source pane as MULTI steps through source code (press `Esc` to abort).

```
while(1){
    S;
    wait;
    E;
    update;
    wait -time 100;
}
```

Example 5. Reading from and Writing to a Variable in Memory

The following script reads from and writes to a variable in memory. You can also use the MULTI commands **memread** and **memwrite** to do this. For information about these commands, see “General Memory Commands” in Chapter 11, “Memory Command Reference” in the *MULTI: Debugging Command Reference* book.

```
> $a = *(unsigned int *)0x8000f0000
> *(unsigned int *)0x8000f0000 = $a + 1
```

Example 6. Performing a toupper() on a String

The following script gets the uppercase version of a string and prints the string to the command pane:

```
define $m_toupper($str) {
    eval $i=0;
    while (( *($str+$i) ) !=0 ) {
        eval ($func_char = (*($str+$i)));
        if (($func_char>='a') && ($func_char<='z')) {
            eval ((*($str+$i)) = $func_char + ('A'-'a'));
        }
        eval $i++;
    }
    mprintf("%s\n", $str);
}
```

Example 7. Advanced: Using Strings in MULTI Scripts

By default, strings that you use are stored in target memory. This can be quite useful when you are debugging because it allows you to do things like make a command line procedure call, passing in as an argument a variable you invent during run time.

However, at times you may want to use a string that does not exist on the target but that you can use in the course of a script or other debug action. To work around the default behavior, you can define a macro that returns a string. Because you cannot actually allocate a string, use the **mprintf** command to simply echo the string to the screen (see the **mprintf** command in Chapter 9, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book). Then use the **substitute** command to make use of the echoed value. The **substitute** command replaces the expression `%EVAL{command}` with the output of *command*. (See the **substitute** command in “Command Manipulation and Macro Commands” in Chapter 16, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.)

Taken together, you have:

```
define ip_addr() {mprintf("192.100.168.2")};  
substitute debugbutton "connect_mpserv" \  
c="connect mpserv %EVAL{ip_addr()}" \  
h="connect to %EVAL{ip_addr()}"
```

This script uses `ip_addr()` as a string variable and creates a Debugger button with two occurrences of `ip_addr()` in it.

Checking the Syntax of Your Script

It can be time consuming to test all the elements of a script by hand. Instead of attempting to do this manually, you can use **MULTI**, which provides a syntax checking feature. This feature validates a script's command syntax without interacting with the target or changing system settings. You can access the syntax checking feature via the **sc** command, which checks your script to ensure it has correct syntax and valid object references. For more information, see "Syntax Checking" in Chapter 12, "Using Expressions, Variables, and Procedure Calls" in the *MULTI: Debugging* book.



Note Use the **BpSyntaxChecking** configuration option to control whether **MULTI** checks the syntax of the commands associated with breakpoints. By default, **MULTI** checks the syntax of a breakpoint's command list when the breakpoint is set and also when you enter the **sc** command. For more information about the **BpSyntaxChecking** configuration option, see Chapter 9, "Configuration Options" in the *MULTI: Editing Files and Configuring the IDE* book.

The **sc** command has three limitations, as follows.

1. When you use the **bu** command to set a breakpoint, the **sc** command cannot check syntax errors in commands associated with the breakpoint. The **bu** command sets up-level breakpoints; the context of the breakpoint depends on dynamic execution. For example, if you enter `bu { print varA }`, **MULTI** cannot determine the up-level procedure until it actually encounters the **bu** command while running the script. When syntax checking the script, **MULTI** has no way to check if the variable `varA` is a valid reference.
2. The **sc** command cannot check for syntax errors in the body of a macro.

3. The **sc** command treats all local variable references that are not located in a breakpoint command as errors. The following script contains an example of this.

```
b main#10 {if (argc>2) {print argc+i;} else {print "Too few arguments"}};  
print argc+i;  
print global_var;
```

If the procedure `main` contains the number variables `argc` and `i`, and there is a global variable `global_var`, the **sc** command accepts the first and the third lines. However, the **sc** command treats the second line as an error because MULTI cannot determine the context in which it performs the `print argc+i` statement.

Running a Script

To run a script, you can:

- Enter the **<** command followed by the *file* argument. For more information, see Chapter 16, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.
- Enter the **-rc** option when starting MULTI from the command line. For more information, see Appendix C, “Command Line Reference” in the *MULTI: Debugging* book.
- Create a button or menu item that uses the **<** command to execute the script. For information, see “Configuring and Customizing Toolbar Buttons” in Chapter 8, “Configuring and Customizing MULTI” in the *MULTI: Editing Files and Configuring the IDE* book.
- Save the script as one of the following types of startup scripts, which MULTI runs automatically at specific times:
 - User script file (**multi.rc**)
 - Program script file (*executable_name.rc*)

For more information about these file types, their locations, and the order in which they run at startup, see “Using Script Files” in Chapter 8, “Configuring and Customizing MULTI” in the *MULTI: Editing Files and Configuring the IDE* book.

- (For board setup scripts) Use the **Connection Editor** or the **connect setup=setup_file** command to specify an **.mbs** file to run before you

download and debug code. For more information, see Chapter 2, “Setting Up Your Target Hardware” in the *MULTI: Configuring Connections* book.

MULTI-Python Scripting

Part II

Introduction to the MULTI-Python Integration

Chapter 3

This Chapter Contains:

- Python Installation
- MULTI-Python Compatibility
- Overview of MULTI-Python Classes
- MULTI-Python Utility Functions
- MULTI-Python Variables
- MULTI-Python Interfaces
- The mpythonrun Utility Program
- Creating a Graphical Interface
- Troubleshooting

The documentation for the MULTI-Python integration assumes that you have a basic knowledge of Python. To learn more about Python, see the Python Web site (<http://www.python.org>).

The MULTI-Python integrated system allows you to use Python code to drive all MULTI IDE components, including the MULTI Project Manager, Debugger, and Editor.

The MULTI-Python integrated system includes the following extensions to standard Python:

- Classes enabling you to access the functions of each MULTI IDE component
- Classes enabling you to access the functions of many MULTI IDE windows
- A general MULTI IDE window class that provides common functions for windows
- Python utility functions and variables for the MULTI IDE
- GUI and non-GUI interfaces in which to run MULTI-Python statements

Python Installation

The MULTI installation package contains a complete Python 2.3.3 installation for your platform. After you have installed MULTI, you should be able to use Python immediately. Alternatively, you may use a customized Python installation. The following sections provide more information about installations.



Note A few Python script files in the Python installation that is shipped with MULTI have been modified to fix bugs.

The Default Python Installation

Python 2.3.3 is located in the top-level directory of your MULTI installation directory.

MULTI-Python integration code is located in the following subdirectory of the MULTI installation directory:

- Windows — **defaults\python**

- UNIX — **defaults/python**

Installing Additional Python Modules (Windows only)

You may want to install additional Python modules to the default Python installation to ease scripting and analysis. Before installing an official Python module, you must register the Python installation in the Windows Registry so that the Python module installer can find it. To register the Python installation, run the **regedit** command to open the Windows Registry and add the following keys:

```
HKEY_LOCAL_MACHINE\Software\Python\PythonCore\2.3\InstallPath
```

```
HKEY_LOCAL_MACHINE\Software\Python\PythonCore\2.3\PythonPath
```

Set each key's value to:

```
multi_install_dir\python
```

Using a Customized Python Installation

You can use a customized Python installation by setting the following environment variables:

- PYTHONHOME — Stores the top-level directory of your Python installation.
- PYTHONPATH — Contains paths to enable searching for Python modules. Paths are kept in the `sys.path` Python variable. You can list them from the standard Python command line interpreter or from the standard Python GUI.

To ensure that the variables take effect, do one of the following:

- In your current MULTI session, set the variables before starting any component of the MULTI IDE.
- If you have already started a component of the MULTI IDE, set the variables and then exit the IDE by selecting **File** → **Exit All** from the Launcher. Then restart the MULTI IDE from the environment (such as an xterm) where the PYTHONHOME and PYTHONPATH environment variables are refreshed.

MULTI-Python Compatibility

MULTI-Python's customized Python interpreter is intended for the Python 2.3.3 release.

At present, MULTI is not compatible with Python 2.4 and later because many changes, including syntax extensions, have been incorporated. For example, the following syntax:

```
from module import (name1, name2)
```

is new in Python 2.4 and is used in basic modules such as **os.py**. MULTI will be integrated with a newer Python release in the future.

Overview of MULTI-Python Classes

GHS_IdeObject Attributes

GHS_IdeObject is the base class of all MULTI-Python service classes, window classes, and Debugger object classes. It contains the following attributes. Many of the attributes refer to *services*. Each MULTI IDE component is usually implemented as a service.

- `serviceName` — Stores the MULTI IDE service name (applicable to some MULTI-Python classes).
- `service` — Stores the MULTI service object (if any).
- `wd` — Stores the working directory of the MULTI service object.
- `maxSecToWaitForNewWindow` — Stores the maximum number of seconds to wait for a MULTI window to appear.
- `checkInterval` — Stores the number of seconds that elapse before Python checks MULTI's status again. By default, the interval between status checks is set to 0.5 seconds, but each component and user can change the interval for each object based on the environment.

The following four GHS_IdeObject attributes are related to command execution. At present, only MULTI Editor commands and MULTI Debugger commands are documented. We suggest that you do not access or directly run the commands of other MULTI services or windows because these

undocumented commands are not guaranteed to be compatible with future MULTI releases.

- `cmdExecStatus` — Stores the command execution status of the corresponding MULTI IDE service or window. Usually, a one (1) indicates success and a zero (0) indicates failure.
- `cmdExecOutput` — Stores the command execution output of the corresponding MULTI IDE service or window.
- `cmdExecObj` — Stores the MULTI-Python object (if any) created by the command execution of the corresponding MULTI IDE service or window.
- `cmdExecPath` — Stores the executed command and indicates how it was executed. This attribute is for debugging purposes.

For `GHS_IdeObject` functions, see “`GHS_IdeObject` Functions” on page 98.

MULTI-Python Service Classes

Each MULTI IDE component is usually implemented as a *service*, and each MULTI service is represented in a Python class. The following table lists the services supported in the MULTI-Python integrated system.

Python Class	MULTI Service	Service Description
<code>GHS_CoBrowse</code>	MULTI Checkout Browser	Provides a graphical, version-controlled checkout.
<code>GHS_Debugger</code>	MULTI Debugger	Provides debugging tools.
<code>GHS_DiffView</code>	MULTI Diff Viewer	Compares two file versions and displays the results in a graphical window.
<code>GHS_Editor</code>	MULTI Editor	Provides text edit functions.
<code>GHS_EventAnalyzer</code>	MULTI EventAnalyzer	Displays and analyzes events.
<code>GHS_Launcher</code>	MULTI Launcher	Allows you to manage MULTI workspaces.
<code>GHS_ProjectManager</code>	MULTI Project Manager	Allows you to manage and build projects.
<code>GHS_PythonGui</code>	MULTI Python GUI	Provides a stand-alone GUI from which you can easily run Python statements.

Python Class	MULTI Service	Service Description
GHS_Terminal	MULTI Serial Terminal	Connects to serial ports and simulates terminals.
GHS_WindowRegister	MULTI Window Register	Tracks all MULTI IDE windows and routes commands to them.

For information about the functions defined for a MULTI-Python class (if any), see Part III.

GHS_Window Attributes

The MULTI-Python class `GHS_Window`, which is derived from the `GHS_IdeObject` class, can represent all MULTI windows.

The following list describes the attributes of class `GHS_Window`:

- `component` — Stores the identifier string for the corresponding Debugger component (only applicable to select Debugger windows).
- `windowName` — Stores the name that is registered for the window. This name may not be the same as the name shown on the window’s title bar.
- `windowId` — Stores the window’s internal ID.
- `winClassName` — Stores the window’s class. For a list of window classes, see “MULTI-Python Window Classes” on page 35.
- `winRegSvcName` — Stores the internal name of the object that registers the window. This attribute is for MULTI’s internal use.
- `service` — Stores the service that contains the window.
- `modalDialogName` — Stores a constant string for the name of all modal dialog boxes. MULTI uses this information internally to identify modal dialog boxes. You should not change the value of this attribute.

For `GHS_Window` functions, see Chapter 8, “Window Functions” and Chapter 9, “Widget Functions”.

MULTI-Python Window Classes

You can launch one or more GUI windows from a MULTI service. Some services, such as the Launcher, launch a window whenever they are created. Other services, such as the Debugger, do not. The former type of service usually shuts down when you close its corresponding GUI window. The Python classes for these services usually have `GHS_Window` as their parent class.

The MULTI-Python integrated system provides Python classes for select MULTI windows. The following table lists these Python classes, which are referred to as *window classes*, and the corresponding MULTI windows.

Python Window Class	Corresponding MULTI Window
<code>GHS_CoBrowseWindow</code>	Checkout Browser
<code>GHS_ConnectionOrganizerWindow</code>	Connection Organizer (used in the Debugger)
<code>GHS_DebuggerWindow</code>	Debugger
<code>GHS_DiffViewWindow</code>	Diff Viewer
<code>GHS_EditorWindow</code>	Editor
<code>GHS_EventAnalyzerWindow</code>	EventAnalyzer
<code>GHS_HelpViewerWindow</code>	Help Viewer
<code>GHS_LauncherWindow</code>	Launcher
<code>GHS_OsaWindow</code>	OSA Explorer (used in the Debugger)
<code>GHS_ProjectManagerWindow</code>	Project Manager
<code>GHS_PyGuiWindow</code>	Python GUI
<code>GHS_TaskManagerWindow</code>	Task Manager (used in the Debugger)
<code>GHS_TerminalWindow</code>	MTerminal
<code>GHS_TraceWindow</code>	Trace List (used in the Debugger)

For information about the functions defined for a MULTI-Python class (if any), see Part III.

MULTI-Python Debugger Object Classes

The MULTI-Python integrated system also provides Python classes for Debugger objects. The following table lists these Python classes.

Python Class	Class Description
GHS_DebuggerApi	Implements general functions for the MULTI Debugger.
GHS_DebugServer	Implements functions for debug server connections.
GHS_Task	Implements functions for debugging tasks or threads in RTOS run-mode debugging environments.

For information about the functions defined for a MULTI-Python class, see Part III.

MULTI-Python Utility Classes

The MULTI-Python integrated system contains utility classes for class GHS_LauncherWindow and for class GHS_Window. The following tables list these utility classes.

For information about the functions defined for a MULTI-Python class, see Part III.

Utility Classes for GHS_LauncherWindow

Python Class	Class Description
GHS_Action	Stores information for an action.
GHS_ActionSequence	Stores information for an action sequence.
GHS_Variable	Stores information for a Launcher variable.
GHS_Workspace	Stores information for a workspace.

Utility Class for GHS_Window

Python Class	Class Description
GHS_MslTree	Stores the content of an MScrollList widget as a parsed tree and provides mechanisms to search for tree nodes, enabling easier access to MULTI MScrollList widgets.

MULTI-Python Miscellaneous Classes

The MULTI-Python integrated system contains the following miscellaneous classes.

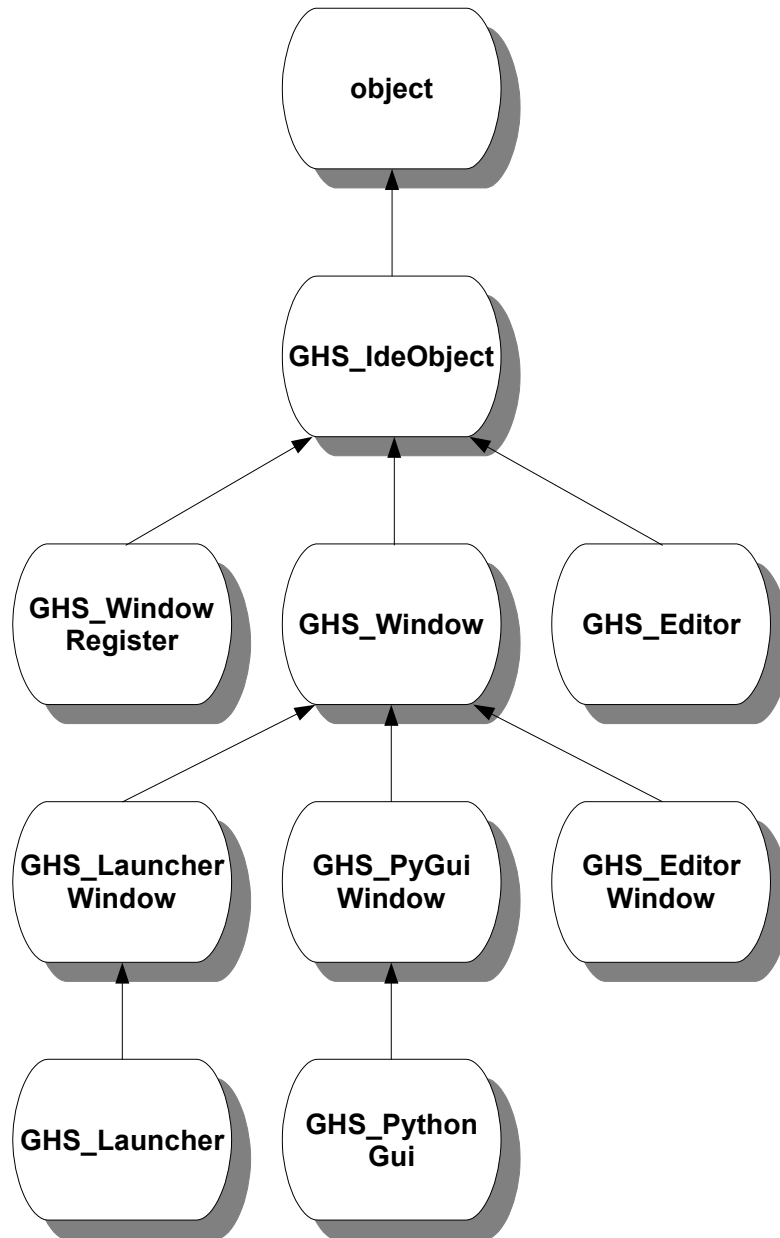
Python Class	Class Description
GHS_AbortExecFile	Aborts file script execution via the utility function <code>GHS_ExecFile()</code> .
GHS_AbortExecOnSignal	Aborts Python execution on signal.
GHS_Exception	Describes MULTI-Python exceptions.
GHS_OsTypes	Stores the IDs of MULTI-supported operating systems.
GHS_TargetIds	Stores the Debugger's target IDs.
GHS_WindowClassNames	Stores MULTI window class names.

For information about the functions defined for a MULTI-Python class, see Part III.

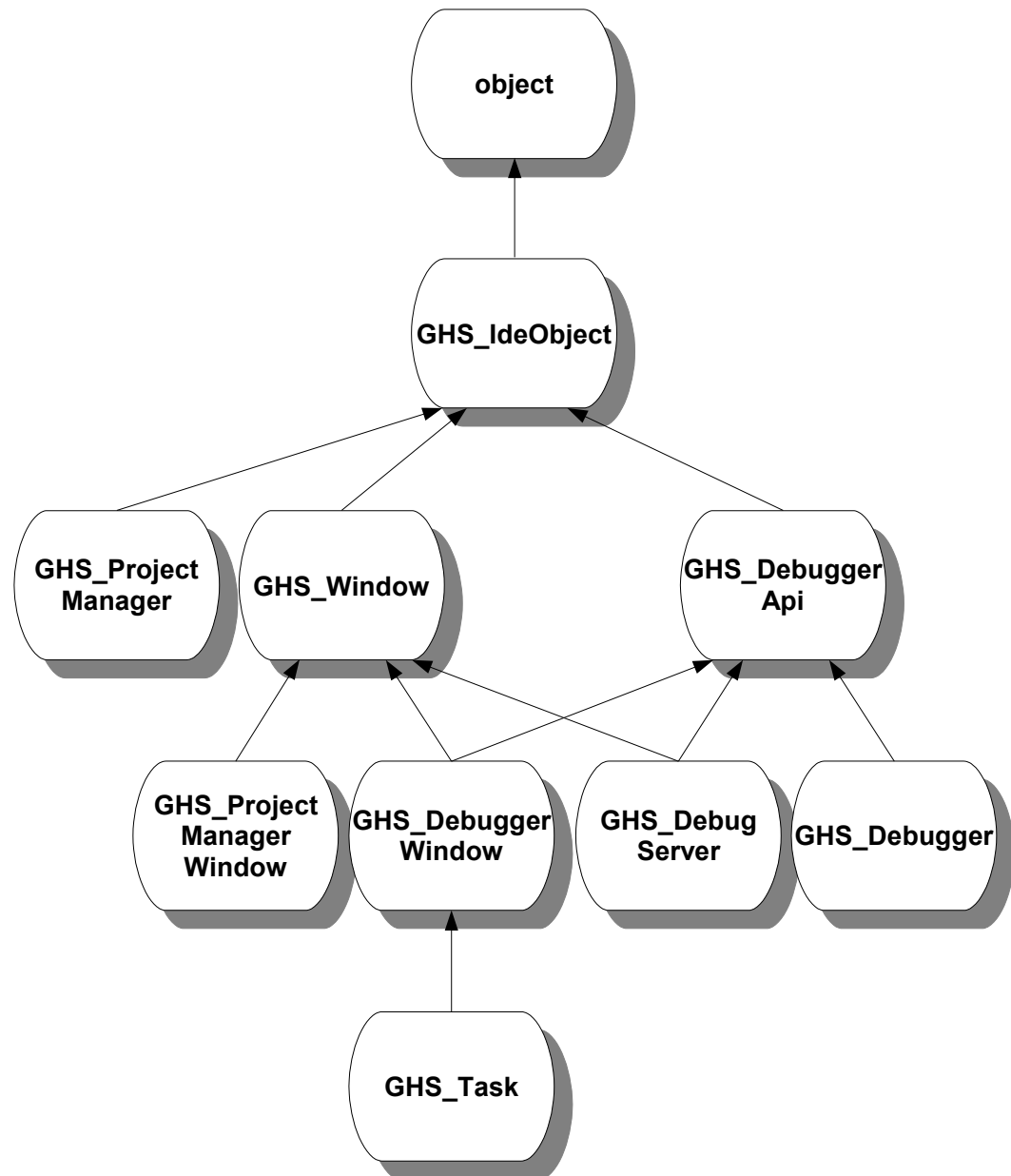
MULTI-Python Class Hierarchy

The following five diagrams represent the MULTI-Python class hierarchy (not all classes are included). Notice that `object`, directly followed by class `GHS_IdeObject`, tops all five hierarchies. The classes are divided into five separate diagrams due to space constraints. Given enough space, they could make up one large diagram. Some class names wrap to a new line because of space constraints. In actuality, class names are always comprised of one continuous text string.

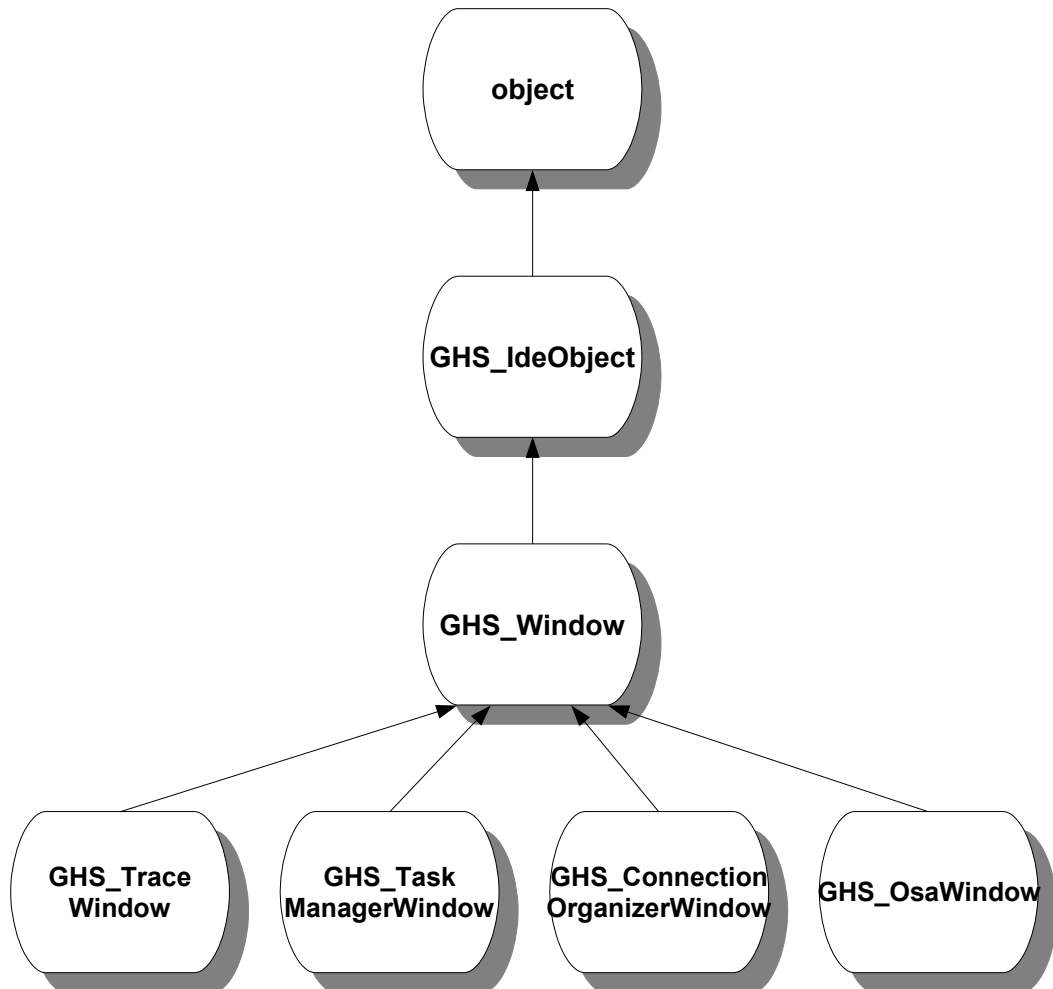
The following diagram includes classes related to MULTI windows, the Editor, the Launcher, and the Python GUI.



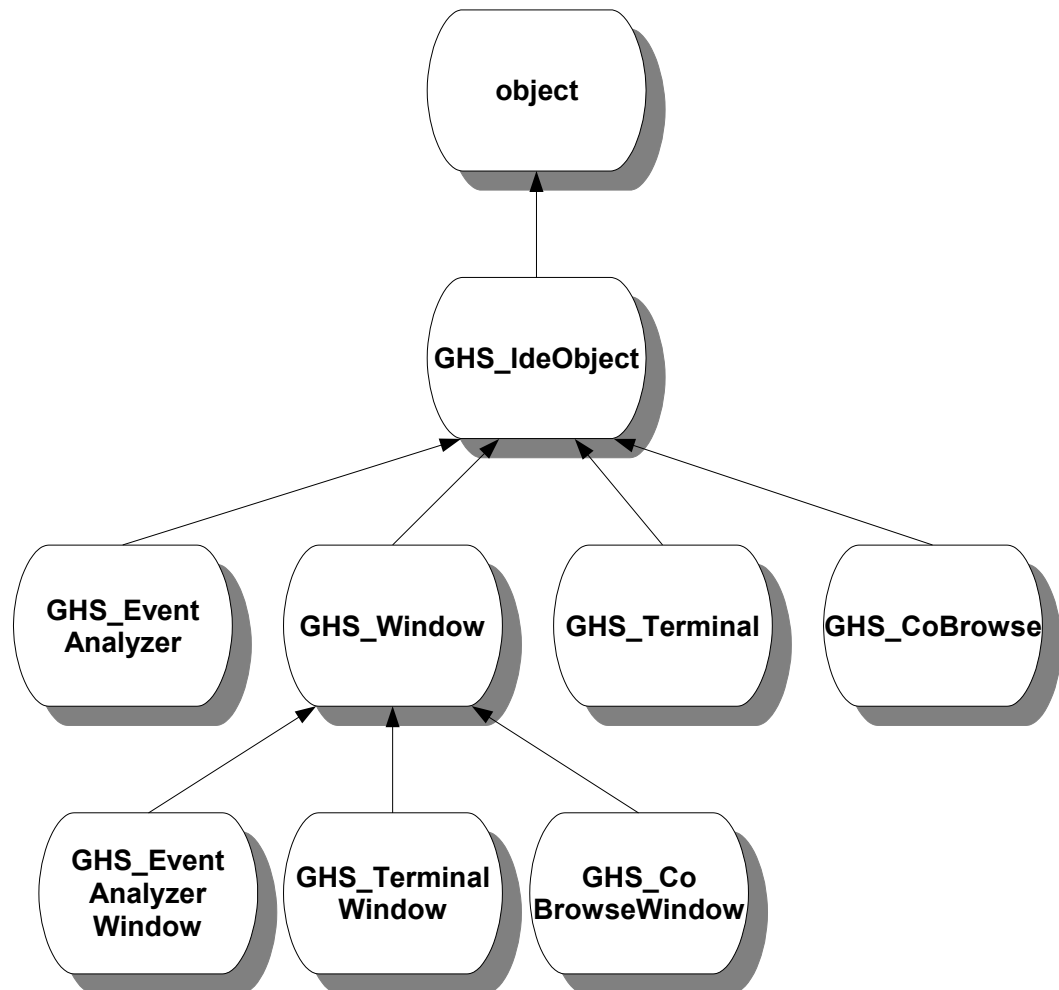
The following diagram includes classes related to the Project Manager, MULTI windows, the Debugger, debug servers, and tasks.



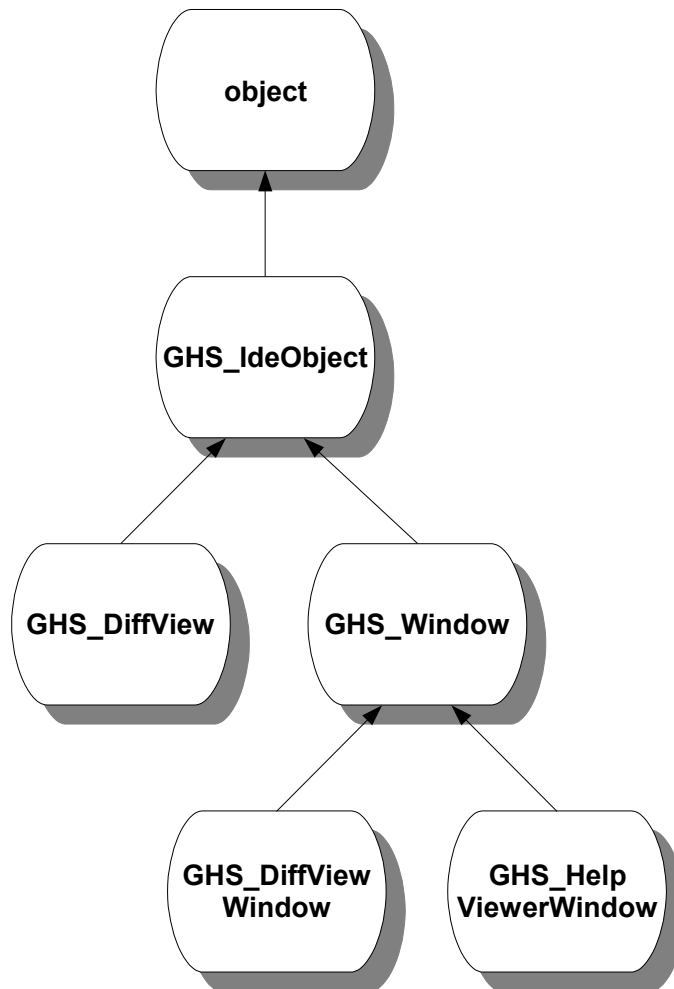
The following diagram includes classes related to MULTI windows, the Trace List, the Task Manager, the Connection Organizer, and the OSA Explorer.



The following diagram includes classes related to the EventAnalyzer, MULTI windows, the Serial Terminal, and the Checkout Browser.



The following diagram includes classes related to the Diff Viewer, MULTI windows, and the Help Viewer.



MULTI-Python Utility Functions

MULTI-Python provides a number of utility functions. You can use these utility functions directly in your Python statements or scripts if you execute them in the MULTI-Python environment. See also “Utility Function Prototypes” on page 92.



Note Utility functions are documented in mixed case for readability, but you may also enter them in *all* lowercase letters. For example, `GHS_ExecFile()` and `ghs_execfile()` are equivalent and are both valid.

- `GHS_ExecFile()` — Runs a Python script just as the `execfile()` built-in Python function does. However, you can raise a `GHS_AbortExecFile()` exception at any place (inside the script file or in one of the nested script files) to abort the execution normally.

When a Python script is executed via the **mpythonrun** utility program or via the MULTI Debugger **python** command, the script is actually run with the `GHS_ExecFile()` function.

- `GHS_PrintObject()` — Prints the given list, tuple or dictionary object in better format than the standard **print** statement. This utility function prints one element per line, indents correctly, etc.
- `GHS_RunShellCommands()` — Executes shell commands, captures their status, grabs output from **stdout** and/or **stderr** on request (grabbing output from **stderr** is supported only in POSIX environments), and returns a tuple such as the following:

```
(exitCode, stdoutOutput, stderrOutput)
```

If `GHS_RunShellCommands()` does not grab the output from **stdout** or **stderr**, the corresponding attributes—`stdoutOutput` and `stderrOutput`—are empty strings (`" "` or `' '`). The corresponding output is displayed on the console (Windows) or in the xterm (UNIX).

- `GHS_System()` — Executes shell commands, grabs output from **stdout** and **stderr** on request, and returns a string containing the requested output. Grabbing output from **stderr** is supported only in POSIX environments. If `GHS_System()` grabs the output from **stderr**, it appends it to the output from **stdout** in the returned string. `GHS_System()` is a simpler version of `GHS_RunShellCommands()`.

MULTI-Python Variables

Pre-Set Variables

The MULTI-Python integrated system includes the following pre-set variables.

- `__ghs_display_height` — Specifies the screen's height in pixels.
- `__ghs_display_width` — Specifies the screen's width in pixels.
- `__ghs_multi_dir` — Specifies the MULTI installation directory.
- `__ghs_python_class_dir` — Specifies the directory containing binary files compiled from MULTI-Python classes.
- `__ghs_site_config_dir` — Specifies the MULTI installation's site configuration directory.
- `__ghs_site_default_dir` — Specifies the MULTI installation's default site configuration directory.
- `__ghs_site_default_python_dir` — Specifies the MULTI-Python integration directory, which resides in the MULTI installation's default site configuration directory.
- `__ghs_user_config_dir` — Specifies your personal configuration directory.
- `__ghs_user_config_python_dir` — Specifies the MULTI-Python integration directory, which resides in your personal configuration directory.
- `__ghs_user_ver_config_dir` — Specifies your personal configuration directory for the current MULTI version.
- `__ghs_user_ver_config_python_dir` — Specifies the MULTI-Python integration directory for the current MULTI version. The directory resides in your personal configuration directory.

Reserved Variable Names

The following table lists object names reserved for particular MULTI-Python classes. You should not assign an arbitrary value to a reserved object name, but you can create an object with the reserved name if one does not already exist.

The third column in the table notes whether the specified object has been pre-created. If an object is pre-created, you can use it immediately in Python

statements or scripts that are executed in the MULTI-Python environment. If an object is not pre-created, its value is `None`.

Python Class	Object Name	Object Description
GHS_Debugger	debugger	Only pre-created in the Py pane and Py Window .
GHS_DebuggerWindow	self_dbw	Only pre-created for the Debugger window where the Py pane is embedded.
GHS_DebugServer	self_dbs	Only pre-created for the debug server associated with the Debugger window. When the Debugger window is not associated with any debug server connection, this variable's value is <code>None</code> . Its value adjusts dynamically when a debug server connection is established or terminated.
GHS_Editor	editor emacs vi	None pre-created by default.
GHS_MemorySpaces	msIds	Always pre-created. This object specifies the commonly used memory spaces supported by the MULTI Debugger.
GHS_ObjectMasks	objMasks	Always pre-created. This object controls whether the projmgr, debugger, editor, emacs, vi, and winreg reserved objects are pre-created by default. You can change the object's value in your customized MULTI-Python integration initialization script.
GHS_OsTypes	osTypes	Always pre-created. This object specifies the operating system IDs recognized by the MULTI Debugger.
GHS_ProjectManager	projmgr	Not pre-created by default.

Python Class	Object Name	Object Description
GHS_TargetIds	targetIds	Always pre-created. This object specifies the target IDs used by the MULTI Debugger.
GHS_WindowClassNames	winClassNames	Always pre-created. This object specifies names for MULTI IDE window classes.
GHS_WindowRegister	winreg	Pre-created by default.

Extending the MULTI-Python Environment

Before a new MULTI-Python context is initialized, the following Python scripts are executed (if they exist) in the order that they are listed below:

1. The Python script kept in the environment variable
BEFORE_GHS_STARTUP_PYTHON
2. The Python script **before_ghs_startup.py** located in the user configuration directory
3. The Python script **before_ghs_startup.py** located in the current working directory

After the MULTI-Python context is initialized, the following Python scripts are executed (if they exist) in the order listed:

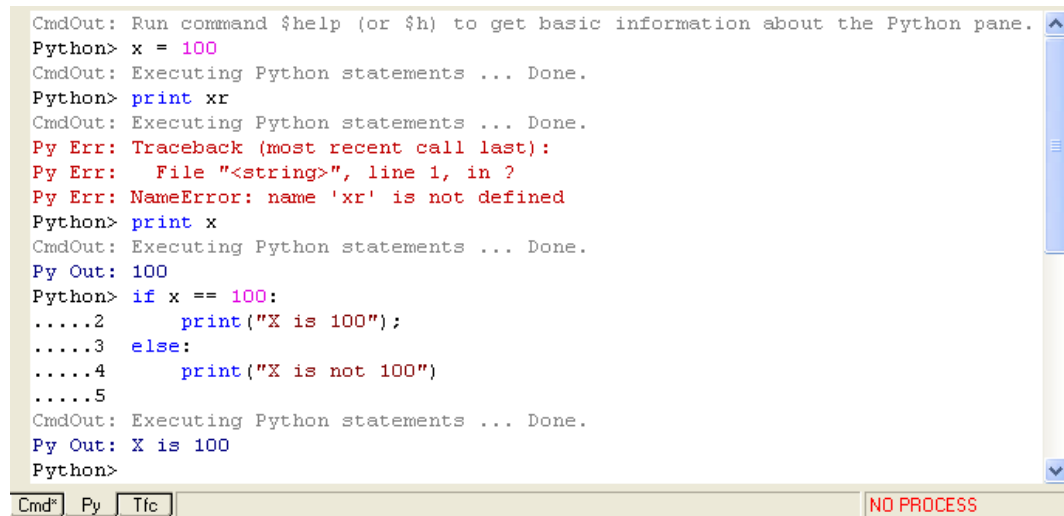
1. The Python script kept in the environment variable
AFTER_GHS_STARTUP_PYTHON
2. The Python script **after_ghs_startup.py** located in the user configuration directory
3. The Python script **after_ghs_startup.py** located in the current working directory

You can also extend the MULTI-Python environment by running Python statements or scripts via the interfaces described in the next section.

MULTI-Python Interfaces

MULTI provides the following interfaces from which you can execute Python statements:

The Debugger's **Py** pane, which you can access by clicking the **Py** tab located at the bottom of the Debugger window:



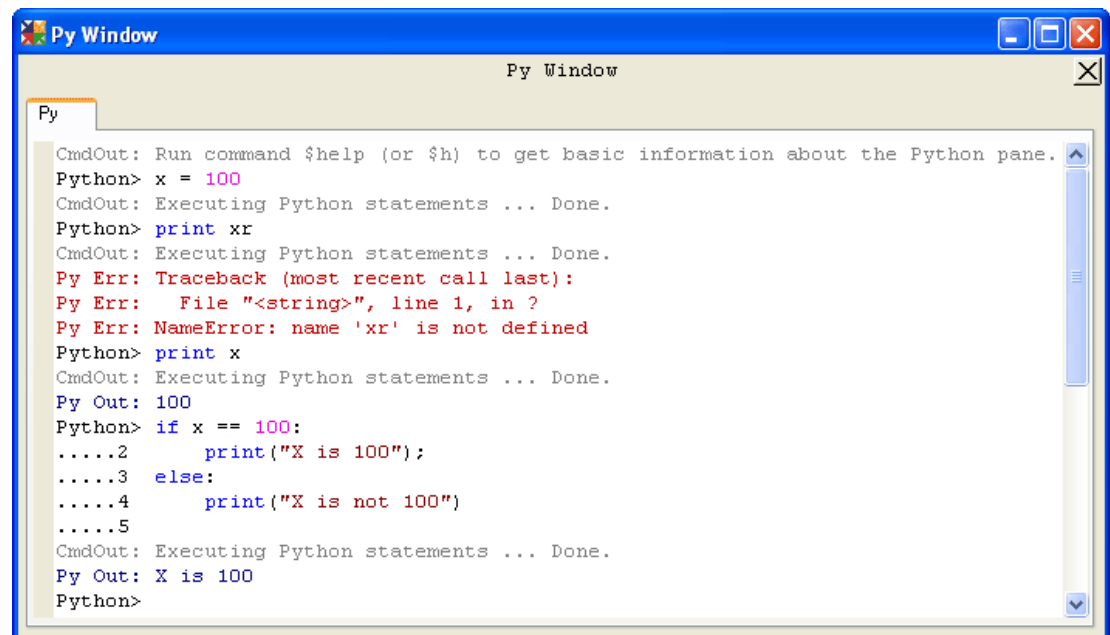
```

CmdOut: Run command $help (or $h) to get basic information about the Python pane.
Python> x = 100
CmdOut: Executing Python statements ... Done.
Python> print xr
CmdOut: Executing Python statements ... Done.
Py Err: Traceback (most recent call last):
Py Err:   File "<string>", line 1, in ?
Py Err: NameError: name 'xr' is not defined
Python> print x
CmdOut: Executing Python statements ... Done.
Py Out: 100
Python> if x == 100:
.....2     print("X is 100");
.....3 else:
.....4     print("X is not 100")
.....5
CmdOut: Executing Python statements ... Done.
Py Out: X is 100
Python>

```

Cmd* Py Tfc NO PROCESS

The **Py Window**, which you can launch by right-clicking in any of the Debugger's panes and selecting **Show Separate Py Window**:

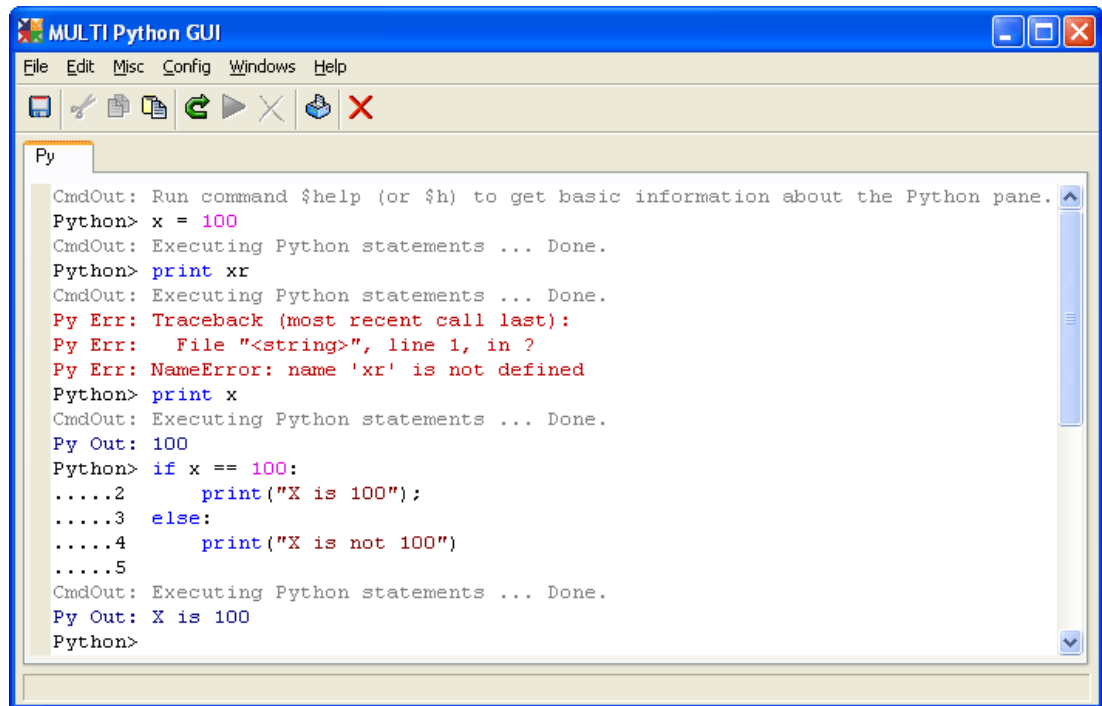


```

Py Window
Py
CmdOut: Run command $help (or $h) to get basic information about the Python pane.
Python> x = 100
CmdOut: Executing Python statements ... Done.
Python> print xr
CmdOut: Executing Python statements ... Done.
Py Err: Traceback (most recent call last):
Py Err:   File "<string>", line 1, in ?
Py Err: NameError: name 'xr' is not defined
Python> print x
CmdOut: Executing Python statements ... Done.
Py Out: 100
Python> if x == 100:
.....2     print("X is 100");
.....3 else:
.....4     print("X is not 100")
.....5
CmdOut: Executing Python statements ... Done.
Py Out: X is 100
Python>

```

The stand-alone **Python GUI**, which you can open from the Launcher by selecting **Components** → **Open Python GUI** or from the host machine's command line by running **mpythongui**:



In addition to executing Python statements via one of the preceding interfaces, you can:

- Run the **python** command in the Debugger command pane. This command can accept a Python statement string or a Python script. For information about the **python** command, see Chapter 16, “Scripting Command Reference” in the *MULTI: Debugging Command Reference* book.
- Add a **Python Statement** or **Python Script** action to your workspace. For information about creating actions, see “Creating or Modifying an Action” in Chapter 5, “Using MULTI Workspaces and Shortcuts” in the *MULTI: Editing Files and Configuring the IDE* book.
- Send Python statements (via your program or a third-party tool such as telnet) to the socket servers provided by the **mpythonrun** program. See “The mpythonrun Utility Program” on page 53.

Interface Comparison

The stand-alone **Python GUI** and the Debugger's **Py** pane and **Py Window** provide similar interfaces where you can run Python statements. A comparison between the **Python GUI** and the Debugger's **Py** pane/**Py Window** follows.



Note The **Py Window** provides more space than the **Py** pane, but they are otherwise the same.

Py Pane & Py Window	Python GUI
Contain a Python pane	Contains a Python pane
Do not contain a menu bar or toolbar	Contains a menu bar and toolbar
Launched from the Debugger	Stand-alone
Begin a new Python history with every Debugger window	Begins a new Python history with every launch
Share Python context, statement history, and output within the same Debugger window	Separate Python context, statement history, and output

The **Py** pane, which is present in the Debugger window, the **Py Window**, and the **Python GUI** provide the following features:

- Syntax coloring and automatic indentation. You can manually adjust indentation by using the keystrokes provided later in this section.
- Line numbers in the command prompt for:
 - Compound statement blocks containing multiple lines.
 - Statements pasted into the **Py** pane.

If an error occurs on one of these lines, the Python interpreter reports the line number so that you can easily find the error location.

- The commands and keyboard shortcuts described in the next section.

Py Pane Commands and Keyboard Shortcuts

The **Py** pane and **Py Window** support the following commands, which are always preceded by the dollar sign (\$). All these commands, with the exception of the **\$verbose** command, are also available to socket server clients started via the **mpythonrun** utility. For more information, see “Starting Socket Servers” on page 56.

The following table lists an equivalent alias with each command, and it lists keyboard shortcuts where applicable.



Note These commands are recognized only when no other text precedes them on the command line. Spaces or tabs preceding the commands are okay.

\$clear [-pane] \$c [-pane] Clears buffered Python statements if you do <i>not</i> specify the <code>-pane</code> option. Clears the content of the Py pane if you do specify the <code>-pane</code> option.
\$display [-new] \$d [-new] Displays Python statements that have already executed. If you specify the <code>-new</code> option, this command displays Python statements that have not yet executed.
\$execute \$e Executes buffered Python statements immediately. By default, this command is bound to: • Ctrl + Enter
\$getinteractive \$gi Prints whether interactive mode is enabled (<code>On</code>) or disabled (<code>Off</code>).

\$help arguments**\$h arguments**

Displays help information in the **Py** pane or in the Help Viewer. The *arguments* that you pass to the command determine where the help appears. For a list of available arguments, run the **\$help** command without any arguments.

By default, this command is bound to:

- Ctrl + h

\$interactive [-on | -off]**\$i [-on | -off]**

Toggles the Python interpreter's support of interactive mode. The optional arguments `-on` and `-off` enable and disable interactive mode, respectively. By default, interactive mode is enabled (`-on`).

In interactive mode, the Python interpreter echoes the value of the object that results from the executed Python statement.

\$restart**\$r**

Restarts the underlying Python interpreter. The old context is discarded.

\$save [filename] [-open]**\$s [filename] [-open]**

Saves buffered Python statements in the file *filename*. If you do not specify a filename, a file chooser prompting you to select a file appears. If you specify the `-open` option, the saved file opens in the MULTI Editor.

With this command, you can easily save Python statements into a file and then replay them later.

By default, **\$save** is bound to:

- Ctrl + s

By default, **\$save -open** is bound to:

- Ctrl + Shift + s

\$verbose [-on | -off]**\$v [-on | -off]**

Toggles verbose mode on and off. The optional arguments `-on` and `-off` enable and disable verbose mode, respectively. By default, it is enabled (`-on`).

In verbose mode, the Python pane prints a message to indicate that it is executing Python statements.

The **Py** pane and **Py Window** support the following keyboard shortcuts, which are bound to the preceding commands.

Ctrl + h
Prints available arguments to the Py pane.
By default, this keyboard shortcut is bound to:
• \$help
Ctrl + i
Indents the defined number of spaces. Indent size is configured in the Options window. For more information about the Indent size option, see “MULTI Editor Configuration Options” in Chapter 9, “Configuration Options” in the <i>MULTI: Editing Files and Configuring the IDE</i> book.
Ctrl + Shift + i
Un-indents the defined number of spaces. Indent size is configured in the Options window. For more information about the Indent size option, see “MULTI Editor Configuration Options” in Chapter 9, “Configuration Options” in the <i>MULTI: Editing Files and Configuring the IDE</i> book.
Ctrl + Enter
Executes buffered Python statements immediately.
By default, this keyboard shortcut is bound to:
• \$execute
Esc
Aborts execution of pending Python statements.
Ctrl + s
Saves buffered Python statements in the file you specify.
By default, this keyboard shortcut is bound to:
• \$save
Ctrl + Shift + s
Saves buffered Python statements in the file you specify and opens the saved file in the MULTI Editor.
By default, this keyboard shortcut is bound to:
• \$save -open

The mpythonrun Utility Program

With the **mpythonrun** utility program, you can run Python statements or scripts from the host operating system's command line and can also start up to two network socket servers. For more information, see "Running Python Statements and Scripts" on page 55 and "Starting Socket Servers" on page 56.

To start **mpythonrun** from the host operating system's command line, enter the path to **mpythonrun.exe**. The **mpythonrun** executable is located in your MULTI installation. For a complete list of **mpythonrun** command line options, see the next section.

mpythonrun Command Line Options

The following table describes available **mpythonrun** command line options.



Note For examples and for more information about many of the following command line options, see "Running Python Statements and Scripts" on page 55 and "Starting Socket Servers" on page 56.

-args <i>Python_script_arguments</i> [-] Specifies Python script arguments. The end of the Python script arguments is indicated either with a dash (-) or by the end of the argument list. For more information, see "Running Python Statements and Scripts" on page 55. This option is valid only when it follows a Python script specification. See also the -script option later in this table.
--

-cr Pads the carriage return (CR) before a new line (LF) when a message is sent to the socket client.
--

-global Allows network-wide socket availability. This is the default behavior. See also the -local option later in this table.
--

-help -h Displays the usage information for mpythonrun.

-local Allows socket availability on the local host only. This option disables connections from other hosts and is useful for security purposes. See also the -global option earlier in this table.
-noconsole Determines where output is sent. If this option is specified before a Python statement or script, mpythonrun prints the output (if any) to <code>stdout</code> or <code>stderr</code> instead of to a console. Otherwise, mpythonrun prints the output (if any) to a console on Windows. If mpythonrun is run as a stand-alone program, this option is not useful. However, if another program calls mpythonrun to run Python scripts and/or statements, and the program must grab the output from mpythonrun , this option is necessary. This option is only valid on Windows and is off by default.
-prompt on off string Turns prompt display on or off, or specifies a customized prompt string. The default prompt string is GHS-Py> .
-script Python_script(s) -f Python_script(s) Specifies the Python script(s) to be run. See also the -args option earlier in this table.
-socket port Specifies the plain socket server. If you do not specify a port number, the system automatically allocates one and prints it out. See also the -telnet option later in this table.
-statement Python_statement(s) -s Python_statement(s) Specifies the Python statement(s) to be run.
-telnet port Specifies the telnet socket server. If you do not specify a port number, the system automatically allocates one and prints it out. See also the -socket option earlier in this table.
-verbose yes no Turns on/off verbose mode. In verbose mode (the default), extra messages are printed to separate output from Python statements or scripts given on the command line.

Running Python Statements and Scripts

To enter command line Python statements with the **mpythonrun** utility program:

- Begin Python statement strings with the **-statement** or **-s** command line option.

For example:

```
mpythonrun -s "import time" -s "print('Current Time:'); print time.asctime()"
```



Note In the MULTI-Python GUI environment, you can abort an executing Python statement by pressing Esc. You cannot abort an executing Python statement in the **mpythonrun** utility program; you can only kill the process. As a result, you should be careful not to submit Python statements containing infinite loops.

To enter command line Python scripts:

- Preface Python scripts with the **-script** or **-f** command line option.
- To transfer arguments to a Python script, begin the argument list with **-args** and end the argument list in one of the following ways:
 - With the end of the argument list (for a single Python script and argument list)
 - With a single dash (-) (for multiple Python scripts and arguments lists or for a single Python script and argument list)

For example, to run a single script and argument list, you could enter:

```
mpythonrun -f pr.py -args haha -a -9 --
```

or

```
mpythonrun -f pr.py -args haha -a -9 -- -
```

To run two scripts, one with an argument list, you could enter:

```
mpythonrun -f pr.py -args haha -a -9 -- - -f pr.py
```

If you specify multiple Python statements or Python script files on the command line, they execute sequentially.

Starting Socket Servers

The **mpythonrun** utility program can start a maximum of two network socket servers. From the command line, you can start a socket server in one of the following ways:

- Plain sockets — Enter the **-socket** command line option followed by a port number. If you do not specify a port number after this option, the system dynamically allocates one and prints it to the console (Windows) or to **stdout** (UNIX).
- Telnet sockets — Enter the **-telnet** command line option followed by a port number. If you do not specify a port number after this option, the system dynamically allocates one and prints it to the console (Windows) or to **stdout** (UNIX).



Note The **mpythonrun** telnet socket server supports well-programmed telnet clients. (A well-programmed telnet client should be able to adjust its settings to work with the telnet socket server.) The Windows telnet client and some others are not well supported. A plain socket is not meant to support telnet protocol, but should work well if your program treats it as a normal socket.

The following example uses a telnet socket to run Python statements:

```
577) telnet hercules 2005
Trying 192.168.101.207...
Connected to hercules.
Escape character is '^]'.
GHS-Py> x = 1
GHS-Py> if x:
.....2     print("x is true");
.....3 else:
.....4     print("x is false");
.....5     print("x = %d" % x);
x is true
x = 1
GHS-Py>
```

If you establish two socket servers at the same time, they share the same Python context. Python statements coming from the two sockets are executed sequentially, but the socket order is not specified. Any Python statements or Python script files that you specify on the command line execute first. The

mpythonrun utility program uses the resulting execution context to execute Python statements from the sockets.

Socket Server Commands

Socket servers accept any Python commands sent to them. All the commands described in “Py Pane Commands and Keyboard Shortcuts” on page 50, with the exception of the **\$verbose** command, are also available to socket servers. In addition to these commands, the following MULTI-Python socket server commands, which are always preceded by the dollar sign (\$), are available.

The following table lists an equivalent alias for each command.



Note These commands are recognized only when no other text precedes them on the command line. Spaces or tabs preceding the commands are okay.

\$prompt [-on | -off | *string*]

\$p [-on | -off | *string*]

Toggles the prompt display, where:

- -on — [default] Enables prompt display.
- -off — Disables prompt display.
- *string* — Changes the prompt to *string* and enables prompt display. The default prompt string is **GHS-Py>**.

If you do not specify any argument, prompt display is toggled.

\$quit

\$q

Shuts down the socket server(s) and quits **mpythonrun**.

Creating a Graphical Interface

The MULTI-Python integrated system provides basic mechanisms that allow for interactive user operations. These mechanisms can ask the user to:

- Give confirmation via an alert dialog box
- Choose between *Yes* and *No*
- Select a value from a given list
- Type a value into a dialog box
- Choose a directory path
- Choose a filename

For examples of these operations, see “Manipulating Windows” on page 62.

Using these mechanisms, you can write an interactive script to drive the MULTI IDE. However, if you are trying to create a complex GUI, the basic mechanisms that MULTI-Python provides may not be sufficient. In this case, you can use a Python GUI package, such as the Tcl/Tk GUI package. For information about this package, see the Python Web site (<http://www.python.org>).

The Python GUI package for Tcl/Tk requires extra third-party tools, including Tcl/Tk and, for Linux on x86, the BLT extension. The MULTI installation contains the following third-party tools for your convenience:

- Windows — Tcl/Tk 8.4 (which is a part of Python’s standard installation on Windows) located at *install_dir\python\tcl*
- Linux on x86 — Tcl/Tk 8.4.11 and BLT 2.4 located at *install_dir/python/tcl_tk*
- Solaris SPARC — Tcl/Tk 8.4.12 located at *install_dir/python/tcl_tk*

You should install any additional required software in your environment.

To view a simple GUI demo that was created with Tcl/Tk, run the following Python statement:

```
execfile(__ghs_site_default_python_dir+os.sep+"ghs_guidemo.py")
```

For more information, see “Using Tcl/Tk to Create a Graphical Interface” on page 85.

Troubleshooting

- The MULTI-Python integrated system does not support Python input functions (such as `input()` and `raw_input()`) or getting input from **stdin**. MULTI-Python provides its own set of functions to handle input. See “GHS_Window Interactive Functions” on page 108.
- If, when you are executing Python statements from the MULTI-Python integration context, you get an error message for the undefined symbol `PyUnicodeUCS2_FromUnicode` or `PyUnicodeUCS4_FromUnicode`, install Python 2.3.3 from the Python Web site (<http://www.python.org>). Follow the Python installation instructions, but give the argument:

`--enable-unicode=ucs2`

or

`--enable-unicode=ucs4`

to the **configure** command.

The MULTI installation’s Python installation is compiled with `ucs4`.

MULTI-Python Tutorials

Chapter 4

This Chapter Contains:

- Manipulating Windows
- Manipulating the Editor
- Manipulating the Debugger
- Using Tcl/Tk to Create a Graphical Interface

The following examples demonstrate how to use MULTI-Python to access the MULTI IDE.

Because these examples were run in the MULTI **Py** pane, the **Python>** prompt is listed before each Python statement.

Manipulating Windows

The following sequential examples demonstrate how to use MULTI-Python to manipulate windows. Many of the examples use the context created by a preceding example. Where this is the case, the appropriate example is referenced.



Note The following are only examples. You may see different results in your MULTI environment.

Example 1. Listing All MULTI IDE Windows

This example lists all the MULTI windows that are currently open.

```
Python> winreg.ShowWins(False)
Py Out: Index      Class Name      Window Name
Py Out: =====      =====
Py Out: 0          Debuggers      mpythonrun.exe
Py Out: 1          Debuggers      me.exe
Py Out: 2          None          Py Window
Py Out: 3          None          References of pySystem
Py Out: 4          Python GUI    MULTI Python GUI
Py Out: 5          Editors      multi_ide_script.xml
Py Out: 6          Editors      ghs_window.py
Py Out:
Py Out: Total number of windows: 7
```

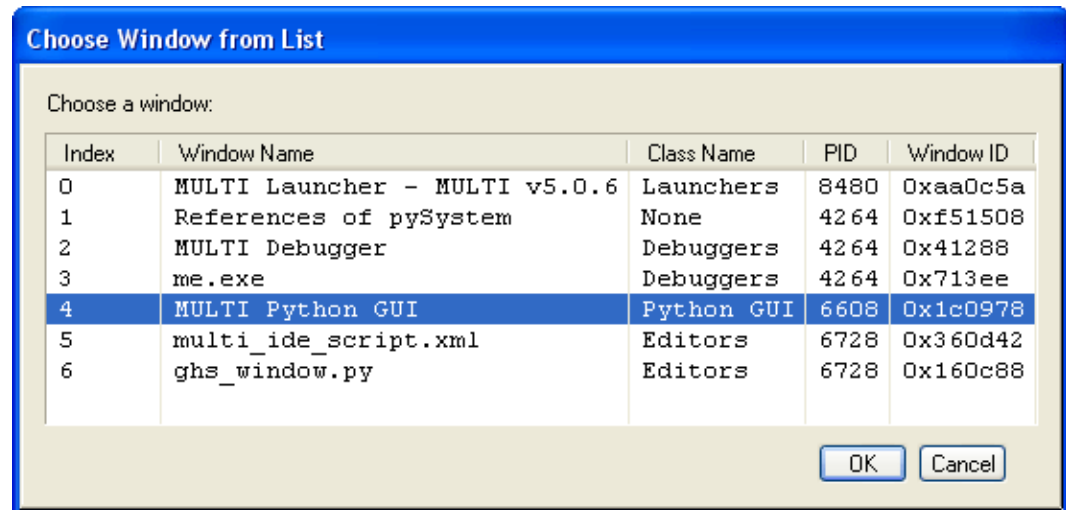
where:

- `winreg` is a pre-created object for class `GHS_WindowRegister`. See “Reserved Variable Names” on page 44.
- `ShowWins()` lists the open windows in the current MULTI IDE session. For more information, see “ShowWindowList()” on page 200.

- The argument `False` that is passed to `ShowWins()` hides the Process ID column that would otherwise appear. For more arguments to `ShowWins()`, see “`ShowWindowList()`” on page 200.

Example 2. Selecting a Window for Display

This example opens a modal dialog box that allows the user to select a window for display. In this example, the stand-alone **Python GUI** was selected.



```
Python> w = winreg.ChooseWin()
Python> print w.windowName
Py Out: MULTI Python GUI
```

where:

- The variable `w` stores the Python object.
- `winreg` is a pre-created object for class `GHS_WindowRegister`. See “Reserved Variable Names” on page 44.
- `ChooseWin()` allows you to choose a window from a window list and then returns an object for the chosen window. For more information, see “`ChooseWindowFromGui()`” on page 203.
- The operation `print w.windowName` prints the name of the returned Python object (which is kept in the variable `w`).
- `MULTI Python GUI` is the name of the window represented by the Python object.

Example 3. Creating a GHS_window Object By Window Index

This example creates a window object for the window with index 5 in Example 1 on page 62.

```
Python> w = winreg.GetWinByIdx(5)
Python> print w.windowName
Py Out: multi_ide_script.xml
```

where:

- The variable `w` stores the Python object.
- `winreg` is a pre-created object for class `GHS_WindowRegister`. See “Reserved Variable Names” on page 44.
- `GetWinByIdx()` gets a `GHS_Window` object from an entry in a window list and returns the created `GHS_Window` object. For more information, see “GetWindowByIndex()” on page 199.
- The argument `5` specifies that the window object is created for the window indexed as 5 in Example 1 on page 62. For more arguments to `GetWinByIdx()`, see “GetWindowByIndex()” on page 199.
- The operation `print w.windowName` prints the name of the returned Python object (which is kept in the variable `w`).
- `multi_ide_script.xml` is the name of the window represented by the Python object, which is an instance of class `GHS_EditorWindow`. Refer to Example 1 on page 62: the window indexed as 5 belongs to the class `Editors`.

Example 4. Creating a GHS_window Object By Window Type

This example creates a `GHS_Window` object for an existing MULTI Debugger window. You can also create `GHS_Window` objects for MULTI Project Manager windows, Editor windows, MTerminal windows, etc. For a complete list of the window types available, see “GHS_WindowRegister Get Window Functions” on page 193.

```
Python> w = winreg.GetDebugger()
Python> print w.windowName
Py Out: mpythonrun.exe
```

where:

- The variable `w` stores the Python object.
- `winreg` is a pre-created object for class `GHS_WindowRegister`. See “Reserved Variable Names” on page 44.
- `GetDebugger()` gets a MULTI Debugger window and returns the created `GHS_DebuggerWindow` object. For more information, see “GetDebuggerWindow()” on page 194.
- The operation `print w.windowName` prints the name of the returned Python object (which is kept in the variable `w`).
- `mpythonrun.exe` is the name of the window represented by the Python object, which is an instance of class `GHS_DebuggerWindow`.

Example 5. Creating a `GHS_Window` Object By Window Name

This example creates a `GHS_Window` object for the MULTI window whose name contains **multi**.

```
Python> w = winreg.GetWin("multi")
Python> print w.windowName
Py Out: multi_ide_script.xml
```

where:

- The variable `w` stores the Python object.
- `winreg` is a pre-created object for class `GHS_WindowRegister`. See “Reserved Variable Names” on page 44.
- `GetWin()` gets a MULTI IDE window from its name, which is a regular expression, and returns the created `GHS_Window` object. For more information, see “GetWindowByName()” on page 199.
- The argument `multi` is the name of the existing window for which the object is created. For more arguments to `GetWin()`, see “GetWindowByName()” on page 199.
- The operation `print w.windowName` prints the name of the returned Python object (which is kept in the variable `w`).
- `multi_ide_script.xml` is the name of the window represented by the Python object, which is an instance of class `GHS_EditorWindow`. Refer to Example 1 on page 62: the window named `multi_ide_script.xml` belongs to the `Editors` class.

Example 6. Bringing a Window to the Foreground

This example brings the specified window to the foreground. Within the context of these sequential examples, the window object `w` specifies the window named **multi_ide_script.xml**. See Example 5 on page 65.

```
Python> w.RestoreWin()
```

where:

- `w` is the Python object created in the preceding example (Example 5 on page 65).
- `RestoreWin()` brings the window to the foreground. For more information, see “RestoreWindow()” on page 137.

Example 7. Moving a Window By a Delta

This example moves the specified window right 100 pixels and down 200 pixels. Within the context of these sequential examples, the window object `w` specifies the window named **multi_ide_script.xml**. See Example 5 on page 65.

```
Python> w.MoveWin(100, 200)
```

where:

- `w` is the Python object created in Example 5 on page 65.
- `MoveWin()` moves the window by a delta or to an absolute position on the current screen. For more information, see “MoveWindow()” on page 135.
- The arguments 100 and 200 specify the number of pixels to move the window. For more arguments to `MoveWin()`, see “MoveWindow()” on page 135.

Example 8. Moving a Window to an Absolute Position

This example positions the left side of the window in the middle of the screen and the top side 100 pixels from the top of the screen. Within the context of these sequential examples, the window object `w` specifies the window named **multi_ide_script.xml**. See Example 5 on page 65.

```
Python> w.MoveWin(__ghs_display_width/2, 100, False)
```

where:

- `w` is the Python object created in Example 5 on page 65.
- `MoveWin()` moves the window by a delta or to an absolute position on the current screen. For more information, see “MoveWindow()” on page 135.
- The argument `__ghs_display_width` is a pre-set MULTI-Python variable that specifies the screen’s width in pixels. See “Pre-Set Variables” on page 44.
- The argument `100` specifies the window’s y coordinate.
- `False` indicates that the preceding two arguments are coordinates within the current screen.

Example 9. Resizing a Window

This example resizes the specified window so that its width is one-third of the screen and its height is one-half of the screen. Within the context of these sequential examples, the window object `w` specifies the window named **multi_ide_script.xml**. See Example 5 on page 65.

```
Python> w.ResizeWin(__ghs_display_width/3,  
.....2          __ghs_display_height/2, False)
```

where:

- `w` is the Python object created in Example 5 on page 65.
- `ResizeWin()` resizes the window by the specified deltas or to the specified dimensions. For more information, see “ResizeWindow()” on page 136.
- The arguments `__ghs_display_width` and `__ghs_display_height` are pre-set MULTI-Python variables that specify the screen’s width and height in pixels. See “Pre-Set Variables” on page 44.
- The argument `False` indicates that the preceding two arguments are absolute dimensions rather than deltas to the window’s existing dimensions.

Example 10. Displaying an Alert Dialog Box

This example displays a modal dialog box that prompts the user for confirmation.



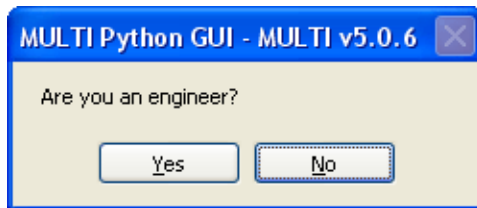
```
Python> w.ShowMsg("We are stuck.\nPress OK to continue.",
.....2             True)
```

where:

- `w` is the Python object created in Example 5 on page 65.
- `ShowMsg()` displays the specified message in the window or in a dialog box. For more information, see “`ShowMessage()`” on page 112.
- The string enclosed in quotation marks is the message that the dialog box displays.
- `True` indicates that the message should be shown in a modal dialog box rather than in the window.

Example 11. Displaying a Yes/No Dialog Box

This example displays a modal dialog box that prompts the user to choose between **Yes** and **No**. In this example, **Yes** was clicked.



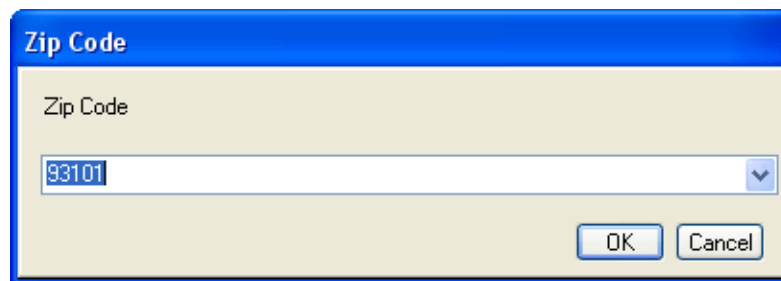
```
Python> ret = w.YesNo("Are you an engineer?")
Python> print ret
Py Out: True
```

where:

- The variable `ret` stores your choice (True for **Yes** and False for **No**).
- `w` is the Python object created in Example 5 on page 65.
- `YesNo()` displays the specified message in a modal dialog box that prompts the user to choose between **Yes** and **No**. This function returns True for **Yes** and False for **No**. For more information, see “ChooseYesNo()” on page 111.
- The string enclosed in quotation marks is the message that the dialog box displays. For more arguments to `YesNo()`, see “ChooseYesNo()” on page 111.
- The operation `print ret` prints the returned value.
- True is the returned value and indicates that the dialog’s **Yes** button was clicked.

Example 12. Displaying an Input Dialog Box

This example displays a modal dialog box that asks for user input. In the dialog box that appears, the user can enter a value or select one from the given list. In this example, **93117** was selected from the list of pre-defined values.



```
Python> ret = w.GetInput("", [93101, 93117, 93118], False,
.....2         "Zip Code")
Python> print ret
Py Out: 93117
```

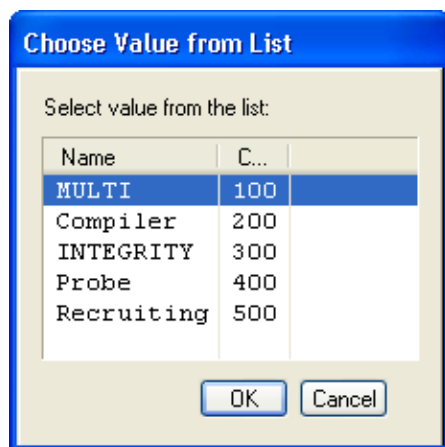
where:

- The variable `ret` stores the string that you choose from the given list or that you type in.
- `w` is the Python object created in Example 5 on page 65.

- `GetInput()` displays the specified values in a modal dialog box and returns user input or an empty string (" ") upon failure or cancellation. For more information, see “`GetInput()`” on page 111.
- The empty string (" ") indicates that the dialog box does not contain a default value.
- 93101, 93117, 93118 specify the dialog box’s pre-defined values.
- `False` indicates that in addition to choosing from the list of pre-defined values, the user can enter their own value.
- `Zip Code` is the prompt that is given in the dialog box. Because no title is specified for the dialog box, the `Zip Code` prompt is also used for the title.
- The operation `print ret` prints the returned value.
- 93117 is the returned value.

Example 13. Displaying a Multiple-Column List

This example displays a modal dialog box that prompts you to choose a value from a two-column list.



When you select a value in a multiple-column list, the value from the first column is returned. In this example, the **Recruiting** entry was selected.

```
Python> ret = w.ChooseFromList(0,
.....2      ["MULTI#100", "Compiler#200", "INTEGRITY#300",
.....3      "Probe#400", "Recruiting#500"],
.....4      "#", ["Name", "Code"])
Python> print ret
Py Out: Recruiting
```


where:

- The variable `ret` stores the string from the first column of the row you choose.
- `w` is the Python object created in Example 5 on page 65.
- `ChooseFromList()` displays the specified list values in a modal dialog box and returns the string selected from the list, or it returns an empty string ("") upon failure or cancellation. For more information, see “ChooseFromList()” on page 110.
- The argument `0` specifies the index of the default value.
- `MULTI#100`, `Compiler#200`, `INTEGRITY#300`, `Probe#400`, and `Recruiting#500` are the pre-defined values present in the dialog box.
- `#` is used as a column separator. For example, for `MULTI#100`, **MULTI** appears in one column and **100** in another.
- `Name` and `Code` are column names.
- The operation `print ret` prints the value chosen in the dialog box.
- `Recruiting` is the returned value.

Example 14. Selecting a File

This example displays a file chooser.

```
Python> ret = w.ChooseFile()  
Python> print ret  
Py Out: /home/puma/dev1/src/indgen.cc
```

where:

- The variable `ret` stores the path for the file you choose.
- `w` is the Python object created in Example 5 on page 65.
- `ChooseFile()` allows you to select a file path via `MULTI`’s file chooser. This function returns the selected file path or an empty string ("") upon failure or cancellation. For more information, see “ChooseFile()” on page 109.
- The operation `print ret` prints the path to the selected file.
- `/home/puma/dev1/src/indgen.cc` is the returned value.

Example 15. Selecting a Directory

This example displays a directory chooser.

```
Python> ret = w.ChooseDir()  
Python> print ret  
Py Out: /home/puma/dev1
```

where:

- The variable `ret` stores the path for the directory you choose.
- `w` is the Python object created in Example 5 on page 65.
- `ChooseDir()` allows you to choose a directory via MULTI's directory chooser. This function returns the selected directory or an empty string (" ") upon failure or cancellation. For more information, see “ChooseDir()” on page 108.
- The operation `print ret` prints the path to the selected directory.
- `/home/puma/dev1` is the returned value.

Example 16. Calling an Interactive Function on winreg

There are a parallel set of interactive, control-related functions in the `GHS_WindowRegister` class (for detailed information, see “GHS_WindowRegister Interactive Functions” on page 201). This example calls an interactive function on the pre-created `winreg` object.

```
Python> ret = winreg.ChooseFromList(0,  
.....2      ["MULTI#100", "Compiler#200", "INTEGRITY#300",  
.....3      "Probe#400", "Recruiting#500"],  
.....4      "#", ["Name", "Code"])  
Python> print ret  
Py Out: Recruiting
```

where:

- The variable `ret` stores the string from the first column of the row you choose.
- `winreg` is a pre-created object for class `GHS_WindowRegister`. See “Reserved Variable Names” on page 44.
- `ChooseFromList()` returns the string selected from the list or returns an empty string (" ") upon failure or cancellation. For information about

the arguments, see Example 13 on page 70. For more information about the function, see “ChooseFromList()” on page 203.

Example 17. Listing a Window’s Widgets

This example lists a window’s widgets. You can use a widget name to view and change the widget’s value. See Example 18 on page 74.

```
Python> w.Widgets()
Py Out: EditMenuBar:      MenuBar
Py Out: :                 Rectangle
Py Out: EditBtns:         ButtonSet
Py Out:   Open:           Button
Py Out:   Save:           Button (dimmed)
Py Out:   -:              Button
Py Out:   Cut:            Button (dimmed)
Py Out:   Copy:           Button (dimmed)
Py Out:   Paste:          Button
Py Out:   Find:           Button
Py Out:   Goto:           Button
Py Out:   -:              Button
Py Out:   Undo:           Button (dimmed)
Py Out:   Redo:           Button (dimmed)
Py Out:   -:              Button
Py Out:   Prev:           Button (dimmed)
Py Out:   Next:           Button (dimmed)
Py Out:   -:              Button
Py Out:   Done:           Button
Py Out:   Close:          Button (dimmed)
Py Out: pulldown:         PullDown (invisible)
Py Out: FilePD:           PullDown
Py Out: ProcPD:           PullDown
Py Out: LineNum:          TextField
Py Out: Status:           Status
Py Out: stReadOnly:       Status
Py Out: stMVC:            Status
Py Out: stMOD:            Status
Py Out: EditPane:         Edit
Py Out: stLnCol:          Status
Py Out: tx_dummy:         Text (invisible)
Py Out: ln_height_errorwin_novis: Line
Py Out: ln_height_errorwin_vis:   Line
Py Out: ov_errorwin:       OmniView (invisible)
Py Out: ln_height_errorwin_curr:  Line
Py Out: sp_errorwin:       Splitter (invisible)
Py Out: adjustForErrorWin: Rectangle
```

where:

- `w` is the Python object created in Example 5 on page 65.
- `Widgets()` dumps information about all widgets in the window. For more information, see “`ShowWidgets()`” on page 149.

Example 18. Displaying the Value of a PullDown Widget

This example displays the value of the PullDown widget that appears in Example 17 on page 73.

```
Python> print w.GetPdVal("pulldown")
Py Out: multi_ide_script.xml
```

where:

- `print` prints out the value of `pulldown`.
- `w` is the Python object created in Example 5 on page 65.
- `GetPdVal()` gets the value of a PullDown widget that is defined in the window and returns a string with the widget’s value. For more information, see “`GetPullDownValue()`” on page 174.
- `pulldown` is the name of the PullDown widget. See Example 17 on page 73.

Example 19. Closing a Window

This example closes the specified window. Within the context of these sequential examples, the window object `w` specifies the window named **multi_ide_script.xml**. See Example 5 on page 65.

```
Python> w.CloseWin()
Python> winreg.ShowWins(False)
Py Out: Index      Class Name      Window Name
Py Out: =====
Py Out: 0          Debuggers      mpythonrun.exe
Py Out: 1          Debuggers      me.exe
Py Out: 2          None          Py Window
Py Out: 3          None          References of pySystem
Py Out: 4          Python GUI     MULTI Python GUI
Py Out: 5          Editors       ghs_window.py
Py Out:
Py Out: Total number of windows: 6
```

where:

- `w` is the Python object created in Example 5 on page 65.
- `CloseWin()` closes the window named **multi_ide_script.xml**. For more information, see “`CloseWindow()`” on page 133.
- `winreg.ShowWins(False)` lists all the MULTI windows that are currently open. (See also Example 1 on page 62.) Note that the window named **multi_ide_script.xml** is not listed.

Manipulating the Editor

The following sequential examples demonstrate how to use MULTI-Python to perform basic operations in the MULTI Editor. Many of the examples use the context created by a preceding example. Where this is the case, the appropriate example is referenced.

Example 20. Creating an Editor Service Object

This example creates an Editor service object.

```
Python> if not editor or not editor.IsAlive():  
.....2      editor = GHS_Editor()
```

For information about specifying the service name and working directory of the Editor service object, see “`__init__()`” on page 266.

Example 21. Opening a File in the Editor

This example opens the specified file (**test.gpy**) in the MULTI Editor.

```
Python> ew = editor.OpenFile("test.gpy")
```

where:

- The variable `ew` stores the Python object.
- `editor` is the object created in Example 20 on page 75.
- `OpenFile()` loads a file into the MULTI Editor and returns a `GHS_EditorWindow` object for the MULTI Editor window. For more information, see “`EditFile()`” on page 267.

- The argument `test.gpy` is the name of the file that the Editor opens. For more arguments to `OpenFile()`, see “`EditFile()`” on page 267.

Example 22. Selecting Text

This example selects text from line 1, column 2 to line 10, column 7 (both line and column numbers start at 1). Within the context of these sequential examples, the Editor object `ew` specifies the **test.gpy** Editor window. See Example 21 on page 75.

```
Python> ew.Select(1, 2, 10, 7)
```

where:

- The variable `ew` is the Python object created in Example 21 on page 75.
- `Select()` selects a range. For more information, see “`SelectRange()`” on page 275.
- 1, 2, 10, and 7 specify the range of selection. For more information, see “`SelectRange()`” on page 275.

Example 23. Copying a Selection

This example copies the existing selection to the clipboard.

```
Python> ew.Copy()
```

where:

- The variable `ew` is the Python object created in Example 21 on page 75.
- `Copy()` copies the selected string to the clipboard. For more information, see “`Copy()`” on page 268.

Example 24. Moving the Cursor

This example moves the cursor to the end of the file.

```
Python> ew.MoveTo(-1, -1)
```

where:

- The variable `ew` is the Python object created in Example 21 on page 75.
- `MoveTo()` moves the cursor to the specified position. For more information, see “`MoveCursor()`” on page 274.

- -1, -1 specifies the end of the file. For more arguments to `MoveTo()`, see “`MoveCursor()`” on page 274.

Example 25. Pasting a Selection

This example pastes the clipboard selection to the current cursor position.

```
Python> ew.Paste()
```

where:

- The variable `ew` is the Python object created in Example 21 on page 75.
- `Paste()` pastes clipboard contents to the cursor’s location. For more information, see “`Paste()`” on page 270.

Example 26. Undoing a Previous Change

This example undoes the last change to the current file. In the context of these sequential examples, the paste operation is undone.

```
Python> ew.Undo()
```

where:

- The variable `ew` is the Python object created in Example 21 on page 75.
- `Undo()` reverses the last change made to the current file. For more information, see “`Undo()`” on page 270.

Manipulating the Debugger

The following sequential examples demonstrate how to use MULTI-Python to access the MULTI Debugger, the debug server, and some debugging windows. Many of the examples use the context created by a preceding example. Where this is the case, the appropriate example is referenced.

Example 27. Creating a Debugger Service Object

This example creates a Debugger service object.

```
Python> if not debugger or not debugger.IsAlive():  
.....2     debugger = GHS_Debugger()
```

For information about specifying the working directory of the Debugger service object, see “`__init__()`” on page 225.

Example 28. Debugging a Program

This example opens the Debugger on an INTEGRITY sim800 BSP kernel.

```
Python> dw = debugger.Debug("/rtos/sim800/kernel")
```

where:

- The variable `dw` stores the Python object for the Debugger window.
- `debugger` is the object created in Example 27 on page 77.
- `Debug()` opens a program in the Debugger. For more information, see “`DebugProgram()`” on page 239.
- The argument `/rtos/sim800/kernel` specifies the path of the program to be debugged. For more arguments to `Debug()`, see “`DebugProgram()`” on page 239.

Example 29. Connecting to a Target

This example connects to **isimppc** from the MULTI Debugger window.

```
Python> fm = dw.Connect("isimppc")
```

where:

- The variable `fm` stores the debug server object.
- The variable `dw` is the Python object created in Example 28 on page 78.
- `Connect()` connects to a target with the specified debug server and returns a `GHS_DebugServer` object for the established debug server connection. For more information, see “`ConnectToTarget()`” on page 214.
- The argument `isimppc` specifies the name of the debug server. For more arguments to `Connect()`, see “`ConnectToTarget()`” on page 214.

Example 30. Running the Program

This example runs the program currently being debugged on the target (here the kernel program).

```
Python> dw.Run()
```


where:

- The variable `dw` is the Python object created in Example 28 on page 78.
- `Run()` runs the program currently being debugged in the Debugger window. For more information, see “Resume()” on page 243.

Example 31. Extracting a Variable’s Value

This example accesses the value of `ASP_Log2PageSize` from the program being debugged on the target.

```
Python> dw.RunCmd('mprintf("%d", ASP_Log2PageSize)', True, False)
Python> valint = int(dw.cmdExecOutput, 0)
```

where:

- The variable `dw` is the Python object created in Example 28 on page 78.
- `RunCmd()` executes a MULTI Debugger command. For more information, see “RunCommands()” on page 250.
- The argument `mprintf("%d", ASP_Log2PageSize)` is the MULTI command executed. See the **mprintf** command in Chapter 9, “Display and Print Command Reference” in the *MULTI: Debugging Command Reference* book. `ASP_Log2PageSize` is a global variable in the INTEGRITY kernel.
- `True` indicates that `RunCmd()` is executed in blocked mode and that MULTI grabs the output (if any).
- `False` specifies that the grabbed output should not be printed.
- The operation `valint = int(dw.cmdExecOutput, 0)`; converts the string value of the `cmdExecOutput` attribute into an integer value kept in the object `valint`.

Example 32. Connecting to a Run-Mode Debug Server

This example establishes a run-mode connection between the local host and the **rtserv** debug server.

```
Python> rm = dw.Rtserv()
```

where:

- The variable `rm` stores the debug server object.

- The variable `dw` is the Python object created in Example 28 on page 78.
- `Rtserv()` connects to an RTOS target with the `rtserve` debug server. For more information, see “ConnectToRtserv()” on page 213.

Example 33. Loading a Module

This example loads the `pizza` demo module to the target.

```
Python> rm.LoadModule("/rtos/sim800/pizza")
```

where:

- The variable `rm` is the debug server object created in Example 32 on page 79.
- `LoadModule()` loads a Dynamic Download module to the target. For more information, see “LoadProgram()” on page 217.
- The argument `/rtos/sim800/pizza` specifies the path of the module to be loaded. For more arguments to `LoadModule()`, see “LoadProgram()” on page 217.

Example 34. Displaying the Task Manager

This example displays the Task Manager.

```
Python> tw = rm.ShowTaskWindow()
```

where:

- The variable `tw` stores the Task Manager object.
- The variable `rm` is the debug server object created in Example 32 on page 79.
- `ShowTaskWindow()` displays the Task Manager (if any) for the debug server. For more information, see “ShowTaskManagerWindow()” on page 218.

Example 35. Selecting the Task Manager’s Flat View

This example changes the Task Manager display from hierarchy view (the default) into flat view. See also Example 36 on page 81.

```
Python> tw.SelectMenu("&View", "F&lat View", True)
```

```
Py Out: You can't switch to a customized group in flat view mode.
```

where:

- The variable `tw` is the Task Manager object created in Example 34 on page 80.
- `SelectMenu()` selects a menu item in the window. For more information, see “`SelectMenu()`” on page 120.
- The arguments `&View` and `F&lat View` specify the Task Manager menu selection that activates flat view. (The **Flat View** menu item toggles between flat view and hierarchy view.) An ampersand (&) must be included before any letter that is underlined in the GUI. For more information about ampersand placement, see “GHS_Window Menu Functions” on page 113.
- `True` indicates that `SelectMenu()` is executed in blocked mode and that `MULTI` grabs Task Manager output and prints it to the **Py** pane. For more arguments to `SelectMenu()`, see “`SelectMenu()`” on page 120.

Example 36. Selecting the Task Manager’s Hierarchy View

This example changes the Task Manager display back to hierarchy view. See also Example 35 on page 80.

```
Python> tw.SelectMenu("&View", "F&lat View", True)
```

where:

- The variable `tw` is the Task Manager object created in Example 34 on page 80.
- `SelectMenu()` selects a menu item in the window. For more information, see “`SelectMenu()`” on page 120.
- The arguments `&View` and `F&lat View` specify the Task Manager menu selection that disables flat view. (The **Flat View** menu item toggles between flat view and hierarchy view.) An ampersand (&) must be included before any letter that is underlined in the GUI. For more information about ampersand placement, see “GHS_Window Menu Functions” on page 113.
- `True` indicates that `SelectMenu()` is executed in blocked mode and that `MULTI` grabs Task Manager output and prints it to the **Py** pane. For more arguments to `SelectMenu()`, see “`SelectMenu()`” on page 120.

Example 37. Dumping the Task List

This example dumps the task list.

```
Python> tw.DumpWidget("Obj_List")
Py Out: MSL: Obj_List
Py Out: -pizzahut
Py Out: >Initial 0x705000 0x01f8/0x1000 Halted 127
Py Out: -information
Py Out: >Initial 0x73f000 0x01f8/0x1000 Halted 127
Py Out: -engineer
Py Out: >Initial 0x779000 0x01f8/0x1000 Halted 127
Py Out: -phonecompany
Py Out: >Initial 0x7b3000 0x01f8/0x1000 Halted 127
Py Out: -kernel
Py Out: >LoaderTask 0x7f0000 0x03e8/0x0800 Pended 254
Py Out: >ResourceManager 0x7f2000 0x0224/0x0600 Pended 254
Py Out: >Idle 0x7f3000 - Running 0
```

where:

- The variable `tw` is the Task Manager object created in Example 34 on page 80.
- `DumpWidget()` dumps the contents of a widget in the window. For more information, see “`DumpWidget()`” on page 147.
- The argument `Obj_List` specifies the name of the widget whose contents will be dumped. For more arguments to `DumpWidget()`, see “`DumpWidget()`” on page 147.

Use `GHS_MslTree` to dump and display a better-formatted task list. In this example, the **Stack** column is hidden so that the output fits onto one page, and the environment is different from that of the previous example, so task IDs do not match those shown above.

```
Python> mt = tw.GetMslTree("Obj_List")
Python> mt.Dump()
Py Out: Row#    Contents of Obj_List (Expansion Name: [Column0] [Column1]...)
Py Out: 0      - pizzahut: [pizzahut] [] [] []
Py Out: 1      |_ Initial: [Initial] [0x705000] [Halted] [127]
Py Out: 2      - information: [information] [] [] []
Py Out: 3      |_ Initial: [Initial] [0x73f000] [Halted] [127]
Py Out: 4      - engineer: [engineer] [] [] []
Py Out: 5      |_ Initial: [Initial] [0x778000] [Halted] [127]
Py Out: 6      - phonecompany: [phonecompany] [] [] []
```

```

Py Out: 7      |_ Initial: [Initial] [0x7b1000] [Halted] [127]
Py Out: 8      - kernel: [kernel] [] [] []
Py Out: 9      | LoaderTask: [LoaderTask] [0x7f0000] [Pended] [254]
Py Out: 10     | ResourceManager: [ResourceManager] [0x7f2000] [Pended] [254]
Py Out: 11     |_ Idle: [Idle] [0x7f3000] [Running] [0]

```

where:

- The variable `mt` is the tree object for the contents of the `Obj_List` widget.

Example 38. Creating a Task Object

This example creates a task object.

```
Python> eng = GHS_Task(rm.component, "engineer", "Initial")
```

where:

- The variable `eng` stores the task object.
- `GHS_Task` implements functions for debugging tasks or threads in RTOS run-mode debugging environments.
- `rm.component` stores the identifier string for the debug server component.
- The argument `engineer` specifies the `AddressSpace`.
- The argument `Initial` specifies the task name.

Example 39. Running a Task Without Attaching

This example runs the task without attaching to it.

```
Python> eng.Run()
```

where:

- The variable `eng` is the task object created in Example 38 on page 83.
- `Run()` runs the task. For more information, see “`Resume()`” on page 262.

Example 40. Attaching to a Task

This example attaches to the task created in Example 38 on page 83.

```

Python> eng.Attach()
Py Out: Target cpu: PowerPC 860 (PowerQUICC)

```

where:

- The variable `eng` is the task object created in Example 38 on page 83.
- `Attach()` attaches to the task. For more information, see “Attach()” on page 260.

Example 41. Setting a Breakpoint

This example sets a breakpoint on the task at `main#10`.

```
Python> eng.SetBp("main#10")
```

where:

- The variable `eng` is the task object created in Example 38 on page 83.
- `SetBp()` sets a software breakpoint at the specified location. For more information, see “SetBreakpoint()” on page 252.
- The argument `main#10` specifies the location where the breakpoint is set. For more arguments to `SetBp()`, see “SetBreakpoint()” on page 252.

Example 42. Terminating a Run-Mode Connection

This example terminates the run-mode connection established in Example 32 on page 79.

```
Python> rm.Disconnect()
```

where:

- The variable `rm` is the debug server object created in Example 32 on page 79.
- `Disconnect()` disconnects the debug server connection. For more information, see “Disconnect()” on page 217.

Example 43. Terminating a Freeze-Mode Connection

This example terminates the freeze-mode connection established in Example 29 on page 78.

```
Python> fm.Disconnect()
```

where:

- The variable `fm` is the debug server object created in Example 29 on page 78.

- `Disconnect()` disconnects the debug server connection. For more information, see “`Disconnect()`” on page 217.

Example 44. Closing the Debugger Window

This example closes the **kernel** Debugger window opened in Example 28 on page 78.

```
Python> dw.CloseWin()
```

where:

- The variable `dw` is the Python object created in Example 28 on page 78.
- `CloseWin()` closes the window. For more information, see “`CloseWindow()`” on page 133.

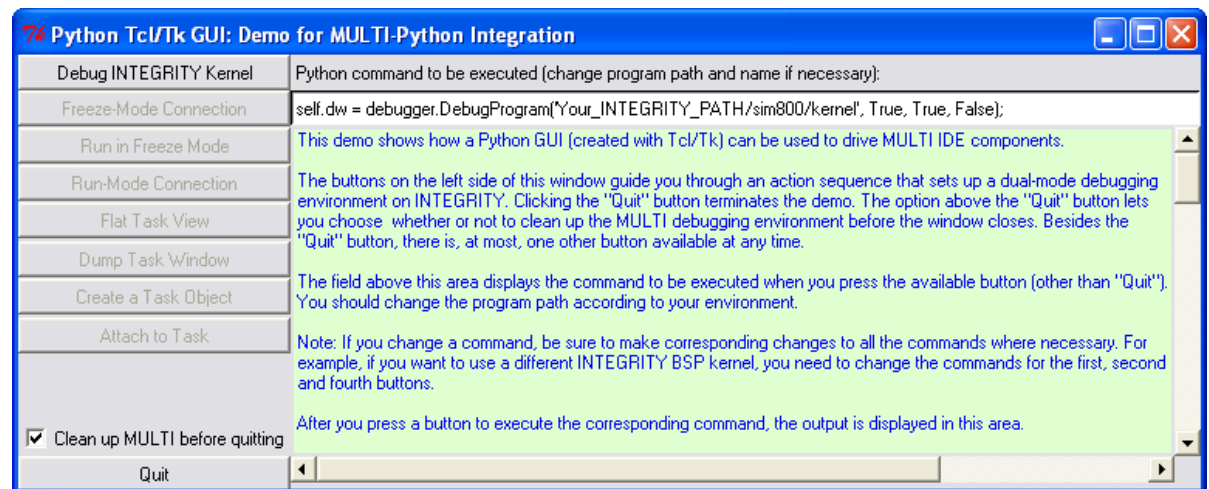
Using Tcl/Tk to Create a Graphical Interface

A simple GUI demo is included with your MULTI installation. The demo shows how a Python GUI that is created with Tcl/Tk can be used to drive MULTI IDE components.

To view the demo, run the following Python statement:

```
execfile(__ghs_site_default_python_dir+os.sep+"ghs_guidemo.py")
```

The resulting demo window contains a number of buttons, a text field for Python statements, and a pane that displays instructions and output.



The window's buttons guide you through an action sequence that sets up a dual-mode debugging environment (freeze mode and run mode) on INTEGRITY. If you do not have an INTEGRITY installation, you can debug and run a stand-alone program instead. Apart from the **Quit** button, one other button (at most) is available at any given time.

The text field displays the Python statement that is executed when you click the button (other than **Quit**) that is available. Some statements require user adjustments, such as a change to a program path or name. If you debug and run a stand-alone program instead of debugging on INTEGRITY, you must change the statements for the first three buttons.

After you click a button to execute a statement, the output is displayed in the window.

To view the demo's source code, run the following Python statements:

```
if not editor: editor = GHS_Editor()
tkew = editor.EditFile(__ghs_site_default_python_dir+
    os.sep+"ghs_guidemo.py")
```

For more information, see “Creating a Graphical Interface” on page 58.

MULTI-Python API Reference

Part III

General Notes on Using Functions

Chapter 5

- Functions and aliases are documented in mixed case, but you may also enter them in *all* lowercase letters (except where it is noted otherwise). For example, `ShowWindowList()` and `showwindowlist()` are equivalent and are both valid.
- Many argument descriptions refer to *blocked mode*. In blocked mode, Python waits for a MULTI operation to finish before continuing on to the next Python statement. In non-blocking mode, Python sends a request to MULTI and then continues immediately to the next statement.

MULTI-Python Utility Function Prototypes

Chapter 6

This Chapter Contains:

- Utility Function Prototypes

This chapter provides information about MULTI-Python utility function prototypes. You can use these utility functions directly in your Python statements or scripts if you execute them in the MULTI-Python environment.

Utility Function Prototypes

The utility function prototypes are listed below. For information about a function, see the page referenced.

- `GHS_ExecFile()` — See page 92.
- `GHS_PrintObject()` — See page 93.
- `GHS_RunShellCommands()` — See page 93.
- `GHS_System()` — See page 94.

GHS_ExecFile()

```
GHS_ExecFile(fileName, gd={}, ld={})
```

Executes the specified Python script file. This function is similar to Python's built-in `execfile()` function. However, as illustrated below, `GHS_ExecFile()` allows you to raise a `GHS_AbortExecFile()` exception at any place (inside the script file or in one of the nested script files) to abort the execution normally.

```
if exitCode != 0:
    raise GHS_AbortExecFile("Exit code: " + str(exitCode));
```

Arguments are:

- `fileName` — Specifies the Python script file to execute.
- `gd` — Specifies the global dictionary that is used as a global namespace to execute the Python script. If empty [default], `GHS_ExecFile()` uses the global dictionary from the MULTI-Python integration environment.
- `ld` — Specifies the local dictionary that is used as a local namespace to execute the Python script. If you omit the local dictionary, it defaults to the `gd` dictionary.

GHS_PrintObject()

```
GHS_PrintObject(obj, printIdx=True)
```

Prints the given list, tuple, or dictionary object in better format than the standard **print** statement. This utility function prints one element per line, indents correctly, etc.

Arguments are:

- **obj** — Specifies the object to print.
- **printIdx** — If **True**, prints element indexes for list, tuple, and dictionary objects. If **False**, does not print element indexes.

The alias is: `GHS_PrintObj()`

GHS_RunShellCommands()

```
GHS_RunShellCommands(commands, grabStdout=True,  
grabStderr=False, printErrMsg=True)
```

Executes the specified shell commands, captures their status, grabs output from **stdout** and/or **stderr** on request (grabbing output from **stderr** is supported only in POSIX environments), and returns a tuple such as the following:

```
(exitCode, stdoutOutput, stderrOutput)
```

If `GHS_RunShellCommands()` does not grab the output from **stdout** or **stderr**, the corresponding attributes—`stdoutOutput` and `stderrOutput`—are empty strings (`" "` or `' '`). The corresponding output is displayed on the console (Windows) or in the xterm (UNIX).

Arguments are:

- **commands** — Specifies the shell commands to execute.
- **grabStdout** — If **True**, grabs output from **stdout** and returns it as a tuple. If **False**, does not grab output from **stdout**, but instead displays it on the console (Windows) or in the xterm (UNIX).
- **grabStderr** — If **True**, grabs output from **stderr** and returns it as a tuple. (This is supported only in POSIX environments.) If **False**,

does not grab output from **stderr**, but instead displays it on the console (Windows) or in the xterm (UNIX).

- `printErrMsg` — If `True`, prints an error message when the exit code is not zero (0). If `False`, does not print an error message when the exit code is not zero (0).

Aliases are: `GHS_RunShellCmds()`, `GHS_ShellCmds()`,
`GHS_Shell()`

GHS_System()

```
GHS_System(commands, grabStdout=True,  
grabStderr=False, printErrMsg=True)
```

Executes the specified shell commands, grabs output from **stdout** and **stderr** on request, and returns a string containing the requested output. If `GHS_System()` does not grab the output from **stdout** or **stderr**, the corresponding output is displayed on the console (Windows) or in the xterm (UNIX).

Grabbing output from **stderr** is supported only in POSIX environments. If `GHS_System()` grabs output from **stderr**, it appends it to the output from **stdout** in the returned string.

This function is a simpler version of `GHS_RunShellCommands()`. This function is also similar to the `system()` function in the `os` module; however, `GHS_System()` is able to grab output.

Arguments are:

- `commands` — Specifies the shell commands to execute.
- `grabStdout` — If `True`, grabs output from **stdout** and returns it as a string. If `False`, does not grab output from **stdout**, but instead displays it on the console (Windows) or in the xterm (UNIX).
- `grabStderr` — If `True`, grabs output from **stderr** and returns it as a string. (This is supported only in POSIX environments.) If `False`, does not grab output from **stderr**, but instead displays it on the console (Windows) or in the xterm (UNIX).

- `printErrMsg` — If `True`, prints an error message when the exit code is not zero (0). If `False`, does not print an error message when the exit code is not zero (0).

Basic Functions

Chapter 7

This Chapter Contains:

- GHS_IdeObject Functions

This chapter documents the `GHS_IdeObject` class, which is the base class of all MULTI-Python service classes, window classes, and Debugger object classes (see “Overview of MULTI-Python Classes” on page 32).

For information about the attributes of class `GHS_IdeObject`, see “GHS_IdeObject Attributes” on page 32.

GHS_IdeObject Functions

The following sections describe functions from class `GHS_IdeObject`.

CleanCmdExecVariables()

```
CleanCmdExecVariables(status=1, output="", obj=None,
cmdPath=" ")
```

Modifies the attributes related to command execution. The default values of the arguments are the default values of the attributes.

Arguments are:

- `status` — Stores the command execution status of the corresponding MULTI IDE service or window. Usually, a one (1) indicates success, and a zero (0) indicates failure.
- `output` — Stores the command execution output of the corresponding MULTI IDE service or window.
- `obj` — Stores the MULTI-Python object (if any) created by the command execution of the corresponding MULTI IDE service or window.
- `cmdPath` — Stores the executed command and indicates how it was executed. This attribute is for debugging purposes.

IsAlive()

```
IsAlive()
```

Checks if the corresponding MULTI IDE service (if any) of a Python object is alive (that is, not down).

The alias is: `Alive()`

Window Functions

Chapter 8

This Chapter Contains:

- GHS_Window Basic Functions
- GHS_Window Configuration Functions
- GHS_Window Directory Functions
- GHS_Window Interactive Functions
- GHS_Window Menu Functions
- GHS_Window Modal Dialog Functions
- GHS_Window Record Functions
- GHS_Window Window Attribute and Manipulation Functions

This chapter documents functions from class `GHS_Window`. Class `GHS_Window` inherits from class `GHS_IdeObject` and implements the general functions of MULTI IDE windows. You can route any command to the corresponding MULTI IDE component via a `GHS_Window` object.

The `GHS_Window` functions are divided into the following sections:

- “`GHS_Window` Basic Functions” on page 101
- “`GHS_Window` Configuration Functions” on page 103
- “`GHS_Window` Directory Functions” on page 105
- “`GHS_Window` Interactive Functions” on page 108
- “`GHS_Window` Menu Functions” on page 113
- “`GHS_Window` Modal Dialog Functions” on page 123
- “`GHS_Window` Window Attribute and Manipulation Functions” on page 133



Note This chapter documents only a subset of the functions from class `GHS_Window`. For information about other `GHS_Window` functions, see Chapter 9, “Widget Functions”.

For information about the attributes of class `GHS_Window`, see “`GHS_Window` Attributes” on page 34.



Note `level` is an argument of some `GHS_Window` functions. This argument specifies the nested level of a modal dialog box and is only applicable if the argument `dialog` is set to `True`. Each MULTI IDE window and dialog box has a name and an internal ID, which MULTI-Python uses to identify the window. Modal dialog boxes are identified by the name `ModalDialog` and by a number that represents their nested level in a sequence of MULTI IDE commands. The most recent modal dialog box has a nested level of zero (0), the second most recent has a nested level of one (1), and so on. MULTI-Python ignores a nested level of zero (0) in a modal dialog box’s name. As a result, `ModalDialog` and `ModalDialog0` are equivalent; both of them identify the most recent modal dialog box.

GHS_Window Basic Functions

The following sections describe basic functions from class `GHS_Window`. For information about the attributes of the `GHS_Window` class, see “`GHS_Window Attributes`” on page 34.

GetCwd()

`GetCwd()`

Gets the current working directory of the window’s process and returns a string containing the directory.

The alias is: `Cwd()`

GetInfo()

`GetInfo(printOut=True)`

Returns a string containing window information.

The argument is:

- `printOut` — If `True`, prints the string. If `False`, does not print the string.

GetPid()

`GetPid()`

Gets the process ID (PID) of the window’s process. This function returns the PID as an integer, or it returns 0 (zero) upon error.

The alias is: `Pid()`

IsSameWindow()

`IsSameWindow(winObj)`

Determines whether the current window object is the same as the specified window object. This function returns `True` if the window objects are the same, and `False` otherwise.

The argument is:

- `winObj` — Specifies a `GHS_Window` object to compare against.

Aliases are: `IsSameWin()`, `SameWin()`

IsWindowAlive()

`IsWindowAlive(clearIfNotAlive=True)`

Determines whether the window for the object still exists and returns `True` if yes, and `False` otherwise.

The argument is:

- `clearIfNotAlive` — If `True` and if the window does not exist, clears window information kept in the window object. If `False`, does not clear window information.

RunCommands()

`RunCommands(cmd, block=True, printOutput=True)`

Runs the specified command(s) in the corresponding MULTI IDE component.

Arguments are:

- `cmd` — Specifies the command(s) to execute. Acceptable commands include documented commands specific to the environment, such as MULTI Editor commands and MULTI Debugger commands. You can directly run these commands with the function.
- `block` — If `True`, executes `RunCommands()` in blocked mode and grabs the output, if any. If `False`, neither executes the function in blocked mode nor grabs the output.

- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

Aliases are: `RunCommand()`, `RunCmd()`, `RunCmds()`

GHS_Window Configuration Functions

The following sections describe the functions from class `GHS_Window` that relate to configuration of the MULTI IDE. For information about MULTI configuration, see Part III, “Configuring the MULTI IDE” in the *MULTI: Editing Files and Configuring the IDE* book.

ClearDefaultConfigFile()

```
ClearDefaultConfigFile(block=True, printOutput=True)
```

Removes MULTI’s default user configuration file. This function returns `True` on success and `False` on failure.

For more information, see “Clearing Configuration Settings” in Chapter 8, “Configuring and Customizing MULTI” in the *MULTI: Editing Files and Configuring the IDE* book.

Arguments are:

- `block` — If `True`, executes `ClearDefaultConfigFile()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

Aliases are: `ClearDftConfigFile()`, `ClearConfigFile()`

LoadConfigFile()

```
LoadConfigFile(fileName="", block=True,  
printOutput=True)
```

Loads the MULTI IDE configuration from a file. This function returns `True` on success and `False` on failure.

Arguments are:

- `fileName` — Specifies the file to be read. If `fileName` is empty, a MULTI file chooser appears so that you can select a file to be read.
- `block` — If `True`, executes `LoadConfigFile()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

SaveConfig()

```
SaveConfig(fileName="", saveToDftFile=False,  
block=True, printOutput=True)
```

Saves the current MULTI IDE configuration to a file and returns `True` on success and `False` on failure.

Arguments are:

- `fileName` — Specifies the file to save to. If `fileName` is an empty string (" ") and `saveToDftFile` is `False` (both are defaults), a MULTI file chooser appears so that you can select a file.
- `saveToDftFile` — If `True`, saves the MULTI IDE configuration to the default configuration file (this is effective only when `fileName` is an empty string). If `False`, you must either specify `fileName` or choose a file from the file chooser. For more information, see “Saving Configuration Settings” in Chapter 8, “Configuring and Customizing MULTI” in the *MULTI: Editing Files and Configuring the IDE* book.
- `block` — If `True`, executes `SaveConfig()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

ShowConfigWindow()

```
ShowConfigWindow(block=True, printOutput=True)
```

Displays the **Options** dialog box, which provides access to many MULTI configuration settings. This function returns `True` on success and `False` on failure. The window object for the **Options** dialog box is kept in the host object's `cmdExecObj` attribute.

Arguments are:

- `block` — If `True`, executes `ShowConfigWindow()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

The alias is: `ShowConfigWin()`

GHS_Window Directory Functions

The following sections describe the directory functions from class `GHS_Window`.

GetIntegrityDistributionDir()

```
GetIntegrityDistributionDir()
```

Gets the INTEGRITY distribution directory and returns a string with the directory or returns an empty string ("") upon failure or cancellation.

Aliases are: `GetIntDistDir()`, `GetIntDir()`

SetIntegrityDistributionDir()

```
SetIntegrityDistributionDir(newDir)
```

Sets the INTEGRITY distribution directory. This function returns a string for the old INTEGRITY distribution directory, or it returns an empty string ("") upon error.

The argument is:

- `newDir` — Specifies the full path to the new INTEGRITY distribution directory.

Aliases are: `SetIntDistDir()`, `SetIntDir()`

GetUvelocityDistributionDir()

`GetUvelocityDistributionDir()`

Gets the u-velOSity distribution directory and returns a string with the directory or returns an empty string (" ") upon failure or cancellation.

Aliases are: `GetUvelDistDir()`, `GetUvelDir()`

SetUvelocityDistributionDir()

`SetUvelocityDistributionDir(newDir)`

Sets the u-velOSity distribution directory. This function returns a string for the old u-velOSity distribution directory, or it returns an empty string (" ") upon error.

The argument is:

- `newDir` — Specifies the full path to the new u-velOSity distribution directory.

Aliases are: `SetUvelDistDir()`, `SetUvelDir()`

GetLatestDir()

`GetLatestDir(dirType="?")`

Gets the latest value for a directory type maintained by the MULTI IDE. This function returns the directory that was most recently used for the specified directory type, or it returns an empty string (" ") upon error.

The argument is:

- `dirType` — Specifies the directory type. Supported directory types are:
 - `BuildFileDir`
 - `ConfigFileDir`
 - `ConnectionFileDir`
 - `EditFileDir`
 - `EventFileDir`
 - `ExecutableDir`
 - `GeneralFileDir`
 - `IntegrateFileDir`
 - `LMAdminFileDir`
 - `MemFilterFileDir`
 - `PlacerFileDir`
 - `TargetFileDir`
 - `TraceFileDir`

The argument `?` specifies any directory type.

The alias is: `GetMruDir()`

SetLatestDir()

```
SetLatestDir(dirType="?", newDir="")
```

Sets the latest value for a directory type maintained by the MULTI IDE. This function returns a string with the existing, most recently used directory of the specified directory type, or it returns an empty string (`" "`) upon error.

Arguments are:

- `dirType` — Specifies the directory type. For a list of supported directory types, see “`GetLatestDir()`” on page 106.
- `newDir` — Specifies a new directory for the directory type.

The alias is: `SetMruDir()`

GHS_Window Interactive Functions

The following sections describe the interactive functions from class `GHS_Window`.

Beep()

```
Beep(count=1, block=False)
```

Beeps the specified number of times and returns `True` on success and `False` on failure.

Arguments are:

- `count` — Specifies the number of beeps.
- `block` — If `True`, executes `Beep()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.

ChooseDir()

```
ChooseDir(dftDir="", prompt="Choose directory:",  
title="Directory Chooser")
```

Allows you to choose a directory via `MULTI`'s directory chooser. This function returns the selected directory, or it returns an empty string ("") upon failure or cancellation.

Arguments are:

- `dftDir` — Specifies the default directory.
- `prompt` — Specifies the prompt string to display in the directory chooser.
- `title` — Specifies the title of the modal directory chooser. If `title` is an empty string (""), the value of `prompt` is used as the title.

The alias is: `DirChooser()`

ChooseFile()

```
ChooseFile(dftFile="", dftDir="", label="OK",  
forOpen=True, existingFile=False, extension="",  
fileTypes="", eraseFilenameWhenDirChange=False,  
title="File Chooser")
```

Allows you to choose a file path via MULTI's file chooser. This function returns the selected file path, or it returns an empty string (" ") upon failure or cancellation.

Arguments are:

- `dftFile` — Specifies the default filename.
- `dftDir` — Specifies the default directory.
- `label` (UNIX only) — Specifies the label of the action button in the file chooser.
- `forOpen` — If `False`, provides file protection by prompting you before allowing you to overwrite an existing file. If `True`, does not prompt you.
- `existingFile` — If `True`, the selected file must already exist. If `False`, the user may create a new file.
- `extension` — Specifies the extension for the selected file.
- `fileTypes` — Specifies the file's MULTI IDE file type.
- `eraseFilenameWhenDirChange` — If `True`, erases the filename located in the file chooser when the directory changes. If `False`, does not erase the filename when the directory changes.
- `title` — Specifies the title of the modal file chooser. If `title` is an empty string (" "), `File Chooser` is used as the title.

The alias is: `FileChooser()`

ChooseFromList()

```
ChooseFromList(dftValueIdx=0, valList=[],  
colValueSep="", colNames=[], prompt="Select value  
from the list:", title="Choose Value from List",  
helpkey="")
```

Allows you to choose a value from a list displayed in a modal dialog box. This function returns the string selected from the list, or it returns an empty string (" ") upon failure or cancellation. New lines are not permitted in string values.

Arguments are:

- `dftValueIdx` — Specifies the index of the list's default value.
- `valList` — Specifies a list of pre-defined values. Each string in `valList` may contain a set of column values separated by `colValueSep`. The returned string is the value of the first column.
- `colValueSep` — Specifies a column-value separator. If `colValueSep` is an empty string (" "), # is used as the separator by default. The separator cannot be a newline character (`\n`).
- `colNames` — Specifies the column names. This should be a list of strings.
- `prompt` — Specifies the prompt string to display.
- `title` — Specifies the title of the modal dialog box. If `title` is an empty string (" "), the value of `prompt` is used as the title.
- `helpkey` — Specifies a string for a MULTI help key.

ChooseWindowFromGui()

```
ChooseWindowFromGui(msg="Choose a window:",  
title="Choose Window from List", wins=None)
```

Allows you to choose a window from a window list displayed in a modal dialog box. This function returns an object for the chosen window, or it returns `None` upon failure or cancellation.

Arguments are:

- `msg` — Specifies the prompt string to display.

- `title` — Specifies the title of the modal dialog box. If `title` is an empty string (""), the value of `msg` is used as the title.
- `wins` — Specifies a list of windows from which you can choose. If you do not specify a window list or if `wins` is an empty string (""), the current MULTI IDE windows in the system are used.

Aliases are: `ChooseWindow()`, `ChooseWin()`

ChooseYesNo()

```
ChooseYesNo(msg, dftChoice=0, printOutput=True)
```

Displays the specified message in a modal dialog box that prompts you to choose between **Yes** and **No**. This function returns `True` for **Yes** and `False` for **No**.

Arguments are:

- `msg` — Specifies the prompt message to display. The message should be a yes/no question.
- `dftChoice` — If 0, the default choice is **No**. If 1, the default choice is **Yes**.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

Aliases are: `YesOrNo()`, `YesNo()`

GetInput()

```
GetInput(dftValue="", valList=[], onlyFromList=False,
prompt="Your input", title="", helpkey="")
```

Gets user input via a modal dialog box. This function returns the input string, or it returns an empty string ("") upon failure or cancellation.

Arguments are:

- `dftValue` — Specifies the default value to return.
- `valList` — Specifies a list of pre-defined values to include in the modal dialog box.

- `onlyFromList` — If `True`, you can only choose from the list of pre-defined values. If `False`, you can choose from the list of pre-defined values, or you can enter your own value.
- `prompt` — Specifies the prompt string to display.
- `title` — Specifies the title of the modal dialog box. If `title` is an empty string (`" "`), the value of `prompt` is used as the title.
- `helpkey` — Specifies a string for a MULTI help key.

ShowMessage()

```
ShowMessage(msg, inDialog=False, error=False,
permanent=False)
```

Displays the specified message in the window or in a dialog box and returns `True` on success and `False` on failure.

Arguments are:

- `msg` — Specifies the message to display.
- `inDialog` — If `True`, displays the message in a modal dialog box. If `False`, displays the message in the window.
- `error` — If `True`, displays the message as an error (if the corresponding window supports the concept). If `False`, does not display the message as an error.
- `permanent` — If `True`, displays the message as a permanent message (if the corresponding window supports the concept). If `False`, does not display the message as a permanent message.

Aliases are: `ShowMsg()`, `DisplayMessage()`, `DisplayMsg()`

Wait()

```
Wait(timeout, local=True, block=False)
```

Blocks the corresponding MULTI IDE component from accepting any commands for the specified amount of time. This function returns `True` on success and `False` on failure.

Arguments are:

- `menu` — Specifies the name of the menu to dump. For more information, see the note at the beginning of “GHS_Window Menu Functions” on page 113.
- `block` — If `True`, executes `DumpMenu()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

DumpMenuBar()

```
DumpMenuBar(block=True, printOutput=True)
```

Dumps the window’s menu bar and returns `True` on success and `False` on failure. The string containing the menu bar is kept in the window object’s `cmdExecOutput` attribute.

Arguments are:

- `block` — If `True`, executes `DumpMenuBar()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

GetCommandToDumpMenu()

```
GetCommandToDumpMenu(menu="", dialog=False, level=0)
```

Gets the command that dumps the specified menu of the window or of a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `menu` — Specifies the name of the menu to dump. For more information, see the note at the beginning of “GHS_Window Menu Functions” on page 113.

- `dialog` — If `True`, specifies that the menu is defined in a modal dialog box. If `False`, specifies that the menu is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 99).

The alias is: `CmdToDumpMenu()`

GetCommandToDumpMenuBar()

```
GetCommandToDumpMenuBar(dialog=False, level=0)
```

Gets the command that dumps the menu bar of the window or of a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `dialog` — If `True`, specifies that the menu bar is located in a modal dialog box. If `False`, specifies that the menu bar is located in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 99).

The alias is: `CmdToDumpMenuBar()`

GetCommandToSelectMenu()

```
GetCommandToSelectMenu(menuName, menuItemName,  
dialog=False, level=0)
```

Gets the command that selects a menu item defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `menuName` — Specifies the menu name.
- `menuItemName` — Specifies the menu item name.

- `dialog` — If `True`, specifies that the menu item is defined in a modal dialog box. If `False`, specifies that the menu item is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 99).

For more information about specifying `menuName` and `menuItemName`, see the note at the beginning of “GHS_Window Menu Functions” on page 113.

The alias is: `CmdToSelMenu()`

GetCommandToSelectMenuPath()

```
GetCommandToSelectMenuPath(menuPath, dialog=False,
                             level=0)
```

Gets the command that selects a menu item defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `menuPath` — Specifies the path (formatted as a Python list) to the menu item. Begin with the main menu and proceed to the menu item. For more information, see the note at the beginning of “GHS_Window Menu Functions” on page 113.
- `dialog` — If `True`, specifies that the menu item is defined in a modal dialog box. If `False`, specifies that the menu item is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 99).

The alias is: `CmdToSelMenuPath()`

GetCommandToSelectSubMenu()

```
GetCommandToSelectSubMenu(menuName, subMenuName,
                           menuItemName, dialog=False, level=0)
```

Gets the command that selects a submenu item defined in the window or in a modal dialog box. This function returns a string for the command. The

command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `menuName` — Specifies the menu name.
- `subMenuName` — Specifies the submenu name.
- `menuItemName` — Specifies the menu item name.
- `dialog` — If `True`, specifies that the submenu item is defined in a modal dialog box. If `False`, specifies that the submenu item is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 99).

For more information about specifying `menuName`, `subMenuName`, and `menuItemName`, see the note at the beginning of “GHS_Window Menu Functions” on page 113.

The alias is: `CmdToSelSubMenu()`

GetCommandToSelectSubSubMenu()

```
GetCommandToSelectSubSubMenu(menuName, subMenuName,  
subSubMenuName, menuItemName, dialog=False, level=0)
```

Gets the command that selects a sub-submenu item defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `menuName` — Specifies the menu name.
- `subMenuName` — Specifies the submenu name.
- `subSubMenuName` — Specifies the sub-submenu name.
- `menuItemName` — Specifies the menu item name.

- `dialog` — If `True`, specifies that the sub-submenu item is defined in a modal dialog box. If `False`, specifies that the sub-submenu item is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 99).

For more information about specifying `menuName`, `subMenuName`, `subSubMenuName` and `menuItemName`, see the note at the beginning of “GHS_Window Menu Functions” on page 113.

The alias is: `CmdToSelSubSubMenu()`

IsMenuItemActive()

`IsMenuItemActive(menuName, menuItemName)`

Determines whether the specified menu item is active (not dimmed) or inactive (dimmed). This function returns `True` if the menu item is active and `False` if it is inactive.

Arguments are:

- `menuName` — Specifies the menu name.
- `menuItemName` — Specifies the menu item name.

For more information about specifying `menuName` and `menuItemName`, see the note at the beginning of “GHS_Window Menu Functions” on page 113.

The alias is: `IsMenuEntryActive()`

IsMenuItemTicked()

`IsMenuItemTicked(menuName, menuItemName)`

Determines whether the specified menu item is ticked. This function returns `True` if the menu item is ticked and `False` if it is not ticked.

Arguments are:

- `menuName` — Specifies the menu name.
- `menuItemName` — Specifies the menu item name.

For more information about specifying `menuName` and `menuItemName`, see the note at the beginning of “GHS_Window Menu Functions” on page 113.

The alias is: `IsMenuEntryTicked()`

IsSubMenuItemActive()

```
IsSubMenuItemActive(menuName, subMenuName,  
menuItemName)
```

Determines whether the specified submenu item is active (not dimmed) or inactive (dimmed). This function returns `True` if the menu item is active and `False` if it is inactive.

Arguments are:

- `menuName` — Specifies the menu name.
- `subMenuName` — Specifies the submenu name.
- `menuItemName` — Specifies the menu item name.

For more information about specifying `menuName`, `subMenuName`, and `menuItemName`, see the note at the beginning of “GHS_Window Menu Functions” on page 113.

The alias is: `IsSubMenuEntryActive()`

IsSubMenuItemTicked()

```
IsSubMenuItemTicked(menuName, subMenuName,  
menuItemName)
```

Determines whether the specified submenu item is ticked. This function returns `True` if the submenu item is ticked and `False` if it is not ticked.

Arguments are:

- `menuName` — Specifies the menu name.
- `subMenuName` — Specifies the submenu name.
- `menuItemName` — Specifies the menu item name.

For more information about specifying `menuName`, `subMenuName`, and `menuItemName`, see the note at the beginning of “GHS_Window Menu Functions” on page 113.

The alias is: `IsSubMenuEntryTicked()`

SelectMenu()

```
SelectMenu(menuName, menuItemName, block=True,
printOutput=True)
```

Selects a menu item defined in the window and returns `True` on success and `False` on failure. The string containing the menu item selection is kept in the window object’s `cmdExecOutput` attribute.

Arguments are:

- `menuName` — Specifies the menu name.
- `menuItemName` — Specifies the menu item name.
- `block` — If `True`, executes `SelectMenu()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

For more information about specifying `menuName` and `menuItemName`, see the note at the beginning of “GHS_Window Menu Functions” on page 113.

Aliases are: `SelMenu()`, `ChooseMenu()`

SelectSubMenu()

```
SelectSubMenu(menuName, subMenuName, menuItemName,
block=True, printOutput=True)
```

Selects a submenu item defined in the window and returns `True` on success and `False` on failure. The string containing the submenu item selection is kept in the window object’s `cmdExecOutput` attribute.

Arguments are:

- `menuName` — Specifies the menu name.
- `subMenuName` — Specifies the submenu name.
- `menuItemName` — Specifies the menu item name.
- `block` — If `True`, executes `SelectSubMenu()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

For more information about specifying `menuName`, `subMenuName`, and `menuItemName`, see the note at the beginning of “GHS_Window Menu Functions” on page 113.

Aliases are: `SelSubMenu()`, `ChooseSubMenu()`

SelectSubSubMenu()

```
SelectSubSubMenu(menuName, subMenuName,  
subSubMenuName, menuItemName, block=True,  
printOutput=True)
```

Selects a sub-submenu item defined in the window and returns `True` on success and `False` on failure. The string containing the menu item selection is kept in the window object’s `cmdExecOutput` attribute.

Arguments are:

- `menuName` — Specifies the menu name.
- `subMenuName` — Specifies the submenu name.
- `subSubMenuName` — Specifies the sub-submenu name.
- `menuItemName` — Specifies the menu item name.
- `block` — If `True`, executes `SelectSubSubMenu()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

For more information about specifying `menuName`, `subMenuName`, `subSubMenuName`, and `menuItemName`, see the note at the beginning of “GHS_Window Menu Functions” on page 113.

Aliases are: `SetSubSubMenu()`, `ChooseSubSubMenu()`

WaitForMenuItem()

```
WaitForMenuItem(menuName, menuItemName, active=True,
duration=-1.0, interval=0.3)
```

Waits for a menu item to attain the specified status: active (not dimmed) or inactive (dimmed). This function returns `True` if the menu item has already attained the specified status or if it attains the specified status before the timeout; it returns `False` otherwise.

Arguments are:

- `menuName` — Specifies the menu name.
- `menuItemName` — Specifies the menu item name.
- `active` — If `True`, active is the awaited status. If `False`, inactive is the awaited status.
- `duration` — Specifies how long the function waits, if at all. The duration may be:
 - A negative number — Indicates that the function waits until the menu item attains the desired status.
 - `0.0` — Indicates that the function does not wait.
 - A positive number — Specifies the maximum number of seconds that the function waits.
- `interval` — Specifies the interval (in seconds) between status checks.

For more information about specifying `menuName` and `menuItemName`, see the note at the beginning of “GHS_Window Menu Functions” on page 113.

Aliases are: `WaitMenuItem()`, `WaitForMenuEntry()`,
`WaitMenuEntry()`

GHS_Window Modal Dialog Functions

To automate certain operations such as dumping the contents of a modal dialog box or changing the values of its widgets, you must register commands before the modal dialog box appears. The functions described in the following sections allow you to register commands. When a modal dialog box is nested to the specified number of levels, the commands are executed.

The most recent modal dialog box has a nested level of zero (0), the second most recent has a nested level of one (1), and so on. If you specify a negative number for the nested level, the commands are executed whenever the MULTI IDE executes modal dialog commands, regardless of the modal dialog nesting level.

Where applicable, the functions in the following sections also allow you to specify a `count`. With the `count` argument, you can register commands to execute a specified number of times (but only once per modal dialog), after which they are automatically removed. By specifying a negative number for the `count`, you can register commands to execute until you explicitly remove them. In this case, the commands are executed whenever the MULTI IDE executes registered modal dialog commands and they are applicable (determined by the modal dialog box nesting level).

GetCommandToRegisterModalDialogCommands()

```
GetCommandToRegisterModalDialogCommands(cmdList,  
level=-1, count=1)
```

Gets the command that registers modal dialog box commands. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `cmdList` — Specifies the list of commands to register.
- `level` — Specifies the modal dialog nesting level at which the commands are executed.
- `count` — Specifies the number of times that the commands are executed.

For more information about the `level` and `count`, see “GHS_Window Modal Dialog Functions” on page 123.

Aliases are: `CmdToRegDlgCmds()`, `CmdToRegDlgCmd()`

RegisterModalDialogCommands()

```
RegisterModalDialogCommands(cmdList, level=-1,  
count=1)
```

Registers commands so that they execute whenever modal dialog boxes are nested to the specified level. This function returns `True` on success and `False` on failure.

Arguments are:

- `cmdList` — Specifies the list of commands to register.
- `level` — Specifies the modal dialog nesting level at which the commands are executed.
- `count` — Specifies the number of times that the commands are executed.

For more information about the `level` and `count`, see “GHS_Window Modal Dialog Functions” on page 123.

Aliases are: `RegDlgCmds()`, `RegDlgCmd()`

RegisterModalDialogToChangePullDownValue()

```
RegisterModalDialogToChangePullDownValue(widgetName,  
value, dlgLevel=0, level=-1, count=1)
```

Registers a command to change the value of a `PullDown` widget defined in a modal dialog box. This function returns `True` on success and `False` on failure.

Arguments are:

- `widgetName` — Specifies the name of the widget.
- `dlgLevel` — Specifies the nested level of the modal dialog box whose `PullDown` widget is changed.
- `level` — Specifies the modal dialog nesting level at which the command is executed.
- `count` — Specifies the number of times that the command is executed.

For more information about the `level` and `count`, see “GHS_Window Modal Dialog Functions” on page 123.

Aliases are: `RegChangePdValue()`, `RegChangePdVal()`

RegisterModalDialogToClickButton()

```
RegisterModalDialogToClickButton(widgetName,  
dlgLevel=0, level=-1, count=1)
```

Registers a command to simulate clicking a `Button` widget defined in a modal dialog box. This function returns `True` on success and `False` on failure.

Arguments are:

- `widgetName` — Specifies the name of the widget.
- `dlgLevel` — Specifies the nested level of the modal dialog box whose `Button` widget is “clicked”.
- `level` — Specifies the modal dialog nesting level at which the command is executed.
- `count` — Specifies the number of times that the command is executed.

For more information about the `level` and `count`, see “GHS_Window Modal Dialog Functions” on page 123.

Aliases are: `RegClickButton()`, `RegClickBut()`

RegisterModalDialogToDoubleClickMslCell()

```
RegisterModalDialogToDoubleClickMslCell(widgetName,  
row, col=0, dlgLevel=0, level=-1, count=1)
```

Registers a command to simulate double-clicking a cell of an `MScrollList` widget defined in a modal dialog box. This function returns `True` on success and `False` on failure.

Arguments are:

- `widgetName` — Specifies the name of the widget.

- `row` — Specifies the index of the row to double-click. The index starts at 0 (zero). For more information, see “DoubleClickMslCell()” on page 157.
- `col` — Specifies the index of the column to double-click. The index starts at 0 (zero).
- `dlgLevel` — Specifies the nested level of the modal dialog box whose `MScrollList` widget is “double-clicked”.
- `level` — Specifies the modal dialog nesting level at which the command is executed.
- `count` — Specifies the number of times that the command is executed.

For more information about the `level` and `count`, see “GHS_Window Modal Dialog Functions” on page 123.

The alias is: `RegDbClickMslCell()`

RegisterModalDialogToDumpWidget()

```
RegisterModalDialogToDumpWidget(widgetName,  
option="", dlgLevel=0, level=-1, count=1)
```

Registers a command to dump the contents of a widget defined in a modal dialog box. This function returns `True` on success and `False` otherwise.

Arguments are:

- `widgetName` — Specifies the name of the widget.
- `option` — Gives more information about what to be dumped if the widget has more items to be dumped. For more information, see “GetCommandToDumpWidget()” on page 147.
- `dlgLevel` — Specifies the nested level of the modal dialog box whose widget contents are dumped.
- `level` — Specifies the modal dialog nesting level at which the command is executed.
- `count` — Specifies the number of times that the command is executed.

For more information about the `level` and `count`, see “GHS_Window Modal Dialog Functions” on page 123.

The alias is: `RegDumpWidget()`

RegisterModalDialogToDumpWindow()

```
RegisterModalDialogToDumpWindow(dlgLevel=0, level=-1,  
count=1)
```

Registers a command to dump the specified modal dialog box. This function returns `True` on success and `False` on failure.

Arguments are:

- `dlgLevel` — Specifies the nested level of the modal dialog box whose contents are dumped.
- `level` — Specifies the modal dialog nesting level at which the command is executed.
- `count` — Specifies the number of times that the command is executed.

For more information about the `level` and `count`, see “GHS_Window Modal Dialog Functions” on page 123.

The alias is: `RegDumpWin()`

RegisterModalDialogToSelectMslCell()

```
RegisterModalDialogToSelectMslCell(widgetName, row,  
col=-1, dlgLevel=0, level=-1, count=1)
```

Registers a command to select cells of an `MScrollList` widget defined in a modal dialog box. This function returns `True` on success and `False` on failure.

Arguments are:

- `widgetName` — Specifies the name of the widget.
- `row` — Specifies the index of the row to select. The index starts at 0 (zero). If the row number is less than 0 and the widget supports single-cell selection, the specified column is selected in all rows.
- `col` — Specifies the index of the column to select. The index starts at 0 (zero). If the column number is less than 0, all columns of the specified row

are selected. If both the row and column numbers are less than 0, all cells of the widget are selected.

- `dlgLevel` — Specifies the nested level of the modal dialog box whose cells are selected.
- `level` — Specifies the modal dialog nesting level at which the command is executed.
- `count` — Specifies the number of times that the command is executed.

For more information about the `level` and `count`, see “GHS_Window Modal Dialog Functions” on page 123.

The alias is: `RegSelMslCell()`

RegisterModalDialogToSelectMslCellByValue()

```
RegisterModalDialogToSelectMslCellByValue(widgetName,  
cellValue, col=0, dlgLevel=0, level=-1, count=1)
```

Registers a command to select a cell of an `MScrollList` widget defined in a modal dialog box. This function returns `True` on success and `False` on failure.

Arguments are:

- `widgetName` — Specifies the name of the widget.
- `cellValue` — Specifies the cell’s value.
- `col` — Specifies the index of the column whose cell values are searched for `cellValue`. The index starts at 0 (zero). The value specified should be a valid column index. A negative value results in selection failure.
- `dlgLevel` — Specifies the nested level of the modal dialog box whose cell is selected.
- `level` — Specifies the modal dialog nesting level at which the command is executed.
- `count` — Specifies the number of times that the command is executed.

For more information about the `level` and `count`, see “GHS_Window Modal Dialog Functions” on page 123.

The alias is: `RegSelMslCellByVal()`

RegisterModalDialogToSelectPullDownMenu()

```
RegisterModalDialogToSelectPullDownMenu(widgetName,  
valIdx, dlgLevel=0, level=-1, count=1)
```

Registers a command to select a menu item of a PullDown widget defined in a modal dialog box. This function returns `True` on success and `False` on failure.

Arguments are:

- `widgetName` — Specifies the name of the widget.
- `valIdx` — Specifies the index of the menu item. The index of the first menu item is 0 (zero).
- `dlgLevel` — Specifies the nested level of the modal dialog box whose menu item is selected.
- `level` — Specifies the modal dialog nesting level at which the command is executed.
- `count` — Specifies the number of times that the command is executed.

For more information about the `level` and `count`, see “GHS_Window Modal Dialog Functions” on page 123.

The alias is: `RegSelPdMenu()`

RegisterModalDialogToShowWidgets()

```
RegisterModalDialogToShowWidgets(dlgLevel=0,  
level=-1, count=1)
```

Registers a command to display the widgets of a modal dialog box. This function returns `True` on success and `False` otherwise.

Arguments are:

- `dlgLevel` — Specifies the nested level of the modal dialog box whose widgets are displayed.
- `level` — Specifies the modal dialog nesting level at which the command is executed.
- `count` — Specifies the number of times that the command is executed.

For more information about the `level` and `count`, see “GHS_Window Modal Dialog Functions” on page 123.

The alias is: `RegShowWidgets()`

RegisterModalDialogToSortMsl()

```
RegisterModalDialogToSortMsl(widgetName, col=0,  
dlgLevel=0, level=-1, count=1)
```

Registers a command to sort an `MScrollList` widget defined in a modal dialog box. This function returns `True` on success and `False` on failure.

Arguments are:

- `widgetName` — Specifies the name of the widget.
- `col` — Specifies the column to sort.
- `dlgLevel` — Specifies the nested level of the modal dialog box whose `MScrollList` widget is sorted.
- `level` — Specifies the modal dialog nesting level at which the command is executed.
- `count` — Specifies the number of times that the command is executed.

For more information about the `level` and `count`, see “GHS_Window Modal Dialog Functions” on page 123.

The alias is: `RegSortMsl()`

RemoveRegisteredModalDialogCommands()

```
RemoveRegisteredModalDialogCommands(level=-1,  
printOutput=True)
```

Removes commands registered at the specified modal dialog nesting level. This function returns `True` on success and `False` on failure.

Arguments are:

- `level` — Specifies the modal dialog nesting level at which commands are registered. If the level is negative, all commands registered to all

modal dialog nesting levels are applicable. For more information, see “GHS_Window Modal Dialog Functions” on page 123.

- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

Aliases are: `RmDlgCmds()`, `RmDlgCmd()`

ShowRegisteredModalDialogCommands()

```
ShowRegisteredModalDialogCommands(level=-1,  
printOutput=True)
```

Displays commands registered at the specified modal dialog nesting level. This function returns `True` on success and `False` on failure.

Arguments are:

- `level` — Specifies the modal dialog nesting level at which commands are registered. If the level is negative, all commands registered to all modal dialog nesting levels are applicable. For more information, see “GHS_Window Modal Dialog Functions” on page 123.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

Aliases are: `ShowDlgCmds()`, `ShowDlgCmd()`

GHS_Window Record Functions

The following section describes the function from class `GHS_Window` that records GUI operations.

RecordGuiOperations()

```
RecordGuiOperations(fileName="", append=False,  
block=True, printOutput=True)
```

Records the Python command equivalents of certain GUI operations executed from the window or from all windows. The commands are recorded to the specified file.

Examples of GUI operations that can be recorded include: selecting a menu from a menu bar, selecting a `Tab` widget, clicking a `Button` or `MScrollList` widget, or modifying a `PullDown` or `TextField` widget. Many other operations, such as selecting a menu option from a right-click menu, or clicking in the Debugger source pane, cannot be recorded.

Arguments are:

- `fileName` — Specifies the file to record to. If `fileName` is not specified, nothing is recorded.
- `append` — If `True`, adds recorded commands after any pre-existing text in the specified file. If `False`, overwrites pre-existing information when recording.
- `block` — If `True`, executes `RecordGuiOperations()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

The alias is: `RecGuiOps()`, `RecGuiOp()`

GHS_Window Window Attribute and Manipulation Functions

The following sections describe the functions from class `GHS_Window` that relate to window attributes and window manipulation.

CloseWindow()

`CloseWindow(block=True, printOutput=True)`

Closes the window and returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `CloseWindow()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

The alias is: `CloseWin()`

GetDimension()

`GetDimension(printOutput=True)`

Gets the window's dimensions. This function returns the dimensions as a tuple such as (width, height), or it returns `None` upon error.

The argument is:

- `printOutput` — If `True`, prints the output, which contains the dimension string. If `False`, does not print the output.

The alias is: `GetDim()`

GetName()

`GetName(printOutput=True)`

Gets the window's registered name. The registered name is the name that MULTI uses internally; it may not be the name shown on the window's title

bar. This function returns the window's name as a string, or it returns an empty string (" ") upon error.

The argument is:

- `printOutput` — If `True`, prints the output, which contains the registered window name. If `False`, does not print the output.

GetPosition()

```
GetPosition(printOutput=True)
```

Gets the window's position. This function returns the position as a tuple such as (x, y), or it returns `None` upon error. The position (0, 0) is in the upper-left corner of the display.

The argument is:

- `printOutput` — If `True`, prints the output containing the position string. If `False`, does not print the output.

The alias is: `GetPos()`

IconifyWindow()

```
IconifyWindow(block=True, printOutput=True)
```

Minimizes the window and returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `IconifyWindow()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

Aliases are: `IconifyWin()`, `IconWindow()`, `IconWin()`, `MinimizeWindow()`, `MinWin()`

IsIconified()

```
IsIconified(printOutput=True)
```

Determines whether the window is minimized and returns `True` if yes, and `False` otherwise.

The argument is:

- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

Aliases are: `IsMinimized()`, `IsMin()`

MoveWindow()

```
MoveWindow(x, y, relative=True)
```

Moves the window by the specified deltas or to the specified position in the display. This function returns `True` on success and `False` on failure.

Arguments are:

- `x` — Specifies the x-delta or the x-coordinate. Whether a delta is specified or a coordinate is specified depends on the argument `relative`.
- `y` — Specifies the y-delta or the y-coordinate. Whether a delta is specified or a coordinate is specified depends on the argument `relative`.
- `relative` — If `True`, indicates that the given values are deltas to the window's current coordinates. If `False`, indicates that the given values are coordinates in the display, where `(0, 0)` is in the upper-left corner.

The alias is: `MoveWin()`

RenameWindow()

```
RenameWindow(winName, winTitle="", iconTitle="")
```

Allows you to rename the window. This function returns `True` on success and `False` on failure.

Arguments are:

- `winName` — Specifies the window's new registered name. The registered name is the name that MULTI uses internally, and it is the name that is displayed in the **Windows** menu of many MULTI tools. The registered name may not be the name shown on the window's title bar.
- `winTitle` — Specifies the name that is displayed in the window's title bar. If you do not specify `winTitle`, the value of `winName` is displayed in the title bar.
- `iconTitle` (UNIX only) — Specifies the name of the window icon. If you do not specify `iconTitle`, the value of `winName` is used for the window icon.

The alias is: `RenameWin`

ResizeWindow()

`ResizeWindow(width, height, relative=True)`

Resizes the window by the specified deltas or to the specified dimensions. This function returns `True` on success and `False` on failure.

Arguments are:

- `width` — Specifies the change in the window's current width or specifies the window's width dimension. Whether a delta is specified or a dimension is specified depends on the argument `relative`.
- `height` — Specifies the change in the window's current height or specifies the window's height dimension. Whether a delta is specified or a dimension is specified depends on the argument `relative`.
- `relative` — If `True`, indicates that the given values are deltas to the window's existing dimensions. If `False`, indicates that the given values are dimensions.

You can use the `__ghs_display_width` and `__ghs_display_height` global variables in the `width` and `height` argument expressions. See "Pre-Set Variables" on page 44.

The alias is: `ResizeWin()`

RestoreWindow()

`RestoreWindow(block=True, printOutput=True)`

Brings the window to the foreground and returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `RestoreWindow()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

Aliases are: `RestoreWin()`, `RaiseWindow()`, `RaiseWin()`

ShowAttributes()

`ShowAttributes(printOutput=True)`

Displays window attributes such as window position, window dimensions, whether the window is minimized or not, and window name. This function returns `True` on success and `False` on failure. The string containing the window attributes is kept in the window object's `cmdExecOutput` attribute.

The argument is:

- `printOutput` — If `True`, prints the output, which contains the attributes. If `False`, does not print the output. Even if the attributes are printed, the corresponding output string is kept in the window object's `cmdExecOutput` attribute.

The alias is: `ShowAttr()`

Widget Functions

Chapter 9

This Chapter Contains:

- GHS_MslTree Attributes and Functions
- GHS_Window Basic Widget Functions
- GHS_Window Button Widget Functions
- GHS_Window Edit and Terminal Widget Functions
- GHS_Window MScrollList Widget Functions
- GHS_Window PullDown Widget Functions
- GHS_Window Tab Widget Functions
- GHS_Window Text Widget Functions
- GHS_Window TextCell Widget Functions
- GHS_Window TextField Widget Functions

This chapter documents functions from the following utility class:

- `GHS_MslTree` — Stores the content of an `MScrollList` widget as a parsed tree and provides mechanisms to search for tree nodes, enabling easier access to `MULTI MScrollList` widgets. You can use the `GetMslTree()` function in class `GHS_Window` (see “`GetMslTree()`” on page 167) to return the tree, which you should not change. Each node in the tree is represented as a `GHS_MslTree` object.

This class inherits from `object` and is a utility class for `GHS_Window`. See “`GHS_MslTree` Attributes and Functions” on page 141.

This chapter also covers functions from class:

- `GHS_Window` — Implements the general functions of `MULTI IDE` windows. You can route any command to the corresponding `MULTI IDE` component via a `GHS_Window` object. This class inherits from class `GHS_IdeObject`.

The `GHS_Window` functions are divided into the following sections:

- “`GHS_Window` Basic Widget Functions” on page 146
- “`GHS_Window` Button Widget Functions” on page 151
- “`GHS_Window` Edit and Terminal Widget Functions” on page 155
- “`GHS_Window` `MScrollList` Widget Functions” on page 156
- “`GHS_Window` `PullDown` Widget Functions” on page 170
- “`GHS_Window` `Tab` Widget Functions” on page 174
- “`GHS_Window` `Text` Widget Functions” on page 180
- “`GHS_Window` `TextCell` Widget Functions” on page 181
- “`GHS_Window` `TextField` Widget Functions” on page 183



Note This chapter documents only a subset of the functions from class `GHS_Window`. For information about other `GHS_Window` functions, see Chapter 8, “Window Functions”.

For information about the attributes of class `GHS_Window`, see “`GHS_Window` Attributes” on page 34.



Note `level` is an argument of some `GHS_Window` functions. This argument specifies the nested level of a modal dialog box and is only applicable if the argument `dialog` is set to `True`. Each MULTI IDE window and dialog box has a name and an internal ID, which MULTI-Python uses to identify the window. Modal dialog boxes are identified by the name `ModalDialog` and by a number that represents their nested level in a sequence of MULTI IDE commands. The most recent modal dialog box has a nested level of zero (0), the second most recent has a nested level of one (1), and so on. MULTI-Python ignores a nested level of zero (0) in a modal dialog box's name. As a result, `ModalDialog` and `ModalDialog0` are equivalent; both of them identify the most recent modal dialog box.

GHS_MslTree Attributes and Functions

Class `GHS_MslTree` stores the content of an `MScrollList` widget as a parsed tree and provides mechanisms to search for tree nodes, enabling easier access to MULTI `MScrollList` widgets. For more information, see the beginning of this chapter (page 139).

For information about the related `MScrollList` widget functions of class `GHS_Window`, see “`GHS_Window MScrollList Widget Functions`” on page 156.

The following list describes the attributes of this class:

- `nodeName` — Stores the name of the node represented by the object. The name is the value of the column (usually column 0) on which the `MScrollList` tree is built.
- `columnValues` — Stores a list of column values for the `MScrollList` widget node represented by the object.
- `parent` — Stores the `GHS_MslTree` object representing the enclosing node in the `MScrollList` tree.
- `depth` — Stores the depth of the node in the `MScrollList` tree.
- `row` — Stores the row index of the node in the `MScrollList` tree.
- `children` — Stores a list of `GHS_MslTree` objects representing the direct children of the node in the `MScrollList` tree.

- `expandMark` — Stores whether the tree node is, or can be, expanded. The `expandMark` attribute may be:
 - "+" — Indicates that the node can be expanded.
 - "-" — Indicates that the node is expanded.
 - " " — Indicates that the node does not expand.

The following sections describe the functions from class `GHS_MslTree`.

DumpTree()

```
DumpTree(recursive=True, treeLine=True,  
printHead=True)
```

Dumps the content of `GHS_MslTree`.

Arguments are:

- `recursive` — If `True`, the dump continues until the operation has no effect. If `False`, this function only dumps the content of the current node.
- `treeLine` — If `True`, lines such as | and _ are printed to show the hierarchy of the dumped information. If `False`, the lines are not printed. For an example, see “`DumpTree()`” on page 291.
- `printHead` — If `True`, prints the header in the dumped content. If `False`, does not print the header.

The alias is: `Dump()`

GetChildrenNumber()

```
GetChildrenNumber(directChildren=False)
```

Gets the number of children (direct or all descendants) of the node.

The argument is:

- `directChildren` — If `True`, only gets the number of direct children of the node. If `False`, gets the number of all descendents of the node.

The alias is: `GetChildNum()`

IsExpandable()

`IsExpandable()`

Determines whether the `GHS_MslTree` node is expandable. This function returns `True` if the node is expandable (that is, if the `MScrollList` tree node contains a + or - sign), and `False` otherwise.

The alias is: `Expandable()`

IsExpanded()

`IsExpanded()`

Determines whether the `GHS_MslTree` node is expanded. This function returns `True` if the node is expanded (that is, if the `MScrollList` tree node contains a - sign), and `False` otherwise.

The alias is: `Expanded()`

IsTopTree()

`IsTopTree()`

Determines whether the `GHS_MslTree` node is the top node. This function returns `True` if the `GHS_MslTree` object represents the abstract top node of the `MScrollList` tree, and `False` otherwise. The abstract top node of the `MScrollList` tree logically contains the nodes in the `MScrollList` as its children.

The alias is: `IsTop()`

SearchByColumnValue()

`SearchByColumnValue(value, column=-1, match=False, all=False, recursive=True)`

Searches by column value for one or more nodes within the current tree (includes the node and its children).

Arguments are:

- `value` — Specifies the column value, which can be a regular expression.
- `column` — Specifies the column to search. If `column` is a negative number, all columns are searched.
- `match` — If `True`, uses Python's regular expression `match()` function to check the specified value. If `False`, uses Python's regular expression `search()` function.
- `all` — If `True`, this function returns a list of all qualified nodes upon success. Upon failure, it returns an empty list. If `False`, this function returns the first qualified node upon success. Upon failure, it returns `None`.
- `recursive` — If `True`, searches all descendants of the `MScrollList` node, if necessary. If `False`, only checks the current node's column values.

The alias is: `SearchByColVal()`

SearchByName()

```
SearchByName(nodeName, match=False, all=False,  
recursive=True)
```

Searches by name for one or more nodes within the current tree (includes the node and its children).

Arguments are:

- `nodeName` — Specifies the name of the node, which can be a regular expression.
- `match` — If `True`, uses Python's regular expression `match()` function to check the specified name. If `False`, uses Python's regular expression `search()` function.
- `all` — If `True`, this function returns a list of all qualified nodes upon success. Upon failure, it returns an empty list. If `False`, this function returns the first qualified node upon success. Upon failure, it returns `None`.
- `recursive` — If `True`, searches all descendants of the `MScrollList` node, if necessary. If `False`, only checks the current node's name.

SearchChildByColumnValue()

```
SearchChildByColumnValue(value, column=-1,  
match=False, all=False, recursive=True)
```

Searches by column value for one or more child nodes of the current tree.

Arguments are:

- **value** — Specifies the column value, which can be a regular expression.
- **column** — Specifies the column to search. If **column** is a negative number, all columns are searched.
- **match** — If **True**, uses Python's regular expression `match()` function to check the specified value. If **False**, uses Python's regular expression `search()` function.
- **all** — If **True**, this function returns a list of all qualified nodes upon success. Upon failure, it returns an empty list. If **False**, this function returns the first qualified node upon success. Upon failure, it returns **None**.
- **recursive** — If **True**, searches all descendants of the `MScrollList` node, if necessary. If **False**, only searches the direct children of the `MScrollList` node.

The alias is: `SearchChildByColVal()`

SearchChildByName()

```
SearchChildByName(nodeName, match=False, all=False,  
recursive=True)
```

Searches by name for one or more child nodes.

Arguments are:

- **nodeName** — Specifies the name of the node, which can be a regular expression.
- **match** — If **True**, uses Python's regular expression `match()` function to check the specified name. If **False**, uses Python's regular expression `search()` function.

- `all` — If `True`, this function returns a list of all qualified nodes upon success. Upon failure, it returns an empty list. If `False`, this function returns the first qualified node upon success. Upon failure, it returns `None`.
- `recursive` — If `True`, searches all descendants of the `MScrollList` node, if necessary. If `False`, only searches the direct children of the `MScrollList` node.

SearchRow()

`SearchRow(row)`

Searches within the current tree (includes the node and its children) for a node on the specified row.

The argument is:

- `row` — Specifies the row.

GHS_Window Basic Widget Functions

The following sections describe the basic widget functions from class `GHS_Window`.



Tip Many of the following functions require you to specify a widget name. You can list widget names with the `ShowWidgets()` function. See “`ShowWidgets()`” on page 149.

DumpAll()

`DumpAll(block=True, printOutput=True)`

Dumps the content of the window and returns `True` on success and `False` on failure. The string containing the window content is kept in the window object’s `cmdExecOutput` attribute.

Arguments are:

- `block` — If `True`, executes `DumpAll()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.

- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

Aliases are: `DumpWindow()`, `DumpWin()`

DumpWidget()

```
DumpWidget(widgetName="", option="", block=True,
printOutput=True)
```

Dumps the content of a widget defined in the window and returns `True` on success and `False` on failure. The string containing the widget's content is kept in the window object's `cmdExecOutput` attribute.

Arguments are:

- `widgetName` — Specifies the name of the widget.
- `option` — Gives additional information about what to dump if the widget has more items to be dumped. The following list gives available options for some widget types:
 - `MScrollList` — selection, highlight, or value
 - `PullDown` — menu or value
 - `TabControl` — selection, content, or value
 - `OmniView` — selection or value
- `block` — If `True`, executes `DumpWidget()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

GetCommandToDumpWidget()

```
GetCommandToDumpWidget(widgetName="", option="",
dialog=False, level=0)
```

Gets the command that dumps the contents of a widget defined in the window or in a modal dialog box. This function returns a string for the command. The

command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the widget.
- `option` — Gives additional information about what to dump if the widget has more items to be dumped. For more information, see “`DumpWidget()`” on page 147.
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

The alias is: `CmdToDumpWidget()`

GetCommandToDumpWindow()

`GetCommandToDumpWindow(dialog=False, level=0)`

Gets the command that dumps the contents of the window or of a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `dialog` — If `True`, gets the command that dumps the contents of a modal dialog box. If `False`, gets the command that dumps the contents of the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

Aliases are: `CmdToDumpWin()`, `CmdToDumpAll()`

GetCommandToShowWidgets()

```
GetCommandToShowWidgets(dialog=False, level=0)
```

Gets the command that displays information about all widgets defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `dialog` — If `True`, gets the command that displays information about a modal dialog box's widgets. If `False`, gets the command that displays information about the window's widgets.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

The alias is: `CmdToShowWidgets()`

ShowWidgets()

```
ShowWidgets(block=True, printOutput=True)
```

Displays information about all widgets defined in the window and returns `True` on success and `False` on failure. The string containing the information is kept in the window object's `cmdExecOutput` attribute.

Arguments are:

- `block` — If `True`, executes `ShowWidgets()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

The following `ShowWidgets()` example lists the widgets of a MULTI Editor window in which a C file is loaded.

```
Python> ew.ShowWidgets()
Py Out: EditMenuBar:      MenuBar
Py Out: :                 Rectangle
Py Out: EditBtns:         ButtonSet
Py Out:   Open:           Button
Py Out:   Save:           Button (dimmed)
Py Out:   -:              Button
Py Out:   Cut:            Button (dimmed)
Py Out:   Copy:           Button (dimmed)
Py Out:   Paste:          Button
Py Out:   Find:           Button
Py Out:   Goto:           Button
Py Out:   -:              Button

Py Out:   Undo:           Button (dimmed)
Py Out:   Redo:           Button (dimmed)
Py Out:   -:              Button
Py Out:   Prev:           Button (dimmed)
Py Out:   Next:           Button (dimmed)
Py Out:   -:              Button
Py Out:   Done:           Button
Py Out:   Close:          Button (dimmed)
Py Out: pulldown:         PullDown (invisible)
Py Out: FilePD:           PullDown
Py Out: ProcPD:           PullDown
Py Out: LineNum:          TextField
Py Out: Status:           Status
Py Out: stReadOnly:       Status
Py Out: stMVC:            Status
Py Out: stMOD:            Status
Py Out: EditPane:         Edit
Py Out: stLnCol:          Status
Py Out: tx_dummy:         Text (invisible)
Py Out: ln_height_errorwin_novis: Line
Py Out: ln_height_errorwin_vis:   Line
Py Out: ov_errorwin:        OmniView (invisible)
Py Out: ln_height_errorwin_curr:  Line
Py Out: sp_errorwin:        Splitter (invisible)
Py Out: adjustForErrorWin:      Rectangle
```

Information about each widget is displayed in the following format:

WidgetName: WidgetType [(ExtraInformation)]

where:

- *WidgetName* — Specifies the widget's name, which is used for `widgetName` in related `GHS_Window` functions. Note that widget names are case-sensitive.
- *WidgetType* — Specifies the widget's type, such as `Button`, `MScrollList`, or `PullDown`.
- *ExtraInformation* — Provides other information (if any), such as whether the widget is invisible, whether a `Button` widget is dimmed, etc.

The alias is: `Widgets()`

GHS_Window Button Widget Functions

The following sections describe the functions from class `GHS_Window` that relate to `Button` widgets.



Tip Many of the following functions require you to specify a widget name. You can list widget names with the `ShowWidgets()` function. See “`ShowWidgets()`” on page 149.

DumpButton()

```
DumpButton(widgetName, block=True, printOutput=True)
```

Dumps the status of a button defined in the window. This function returns `True` on success and `False` otherwise.

Arguments are:

- `widgetName` — Specifies the name of the `Button` widget.
- `block` — If `True`, executes `DumpButton()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

The alias is: `DumpBut()`

GetCommandToClickButton()

```
GetCommandToClickButton(widgetName, dialog=False,  
level=0)
```

Gets the command that simulates clicking a button. The button may be defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the `Button` widget.
- `dialog` — If `True`, specifies that the button is defined in a modal dialog box. If `False`, specifies that the button is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

Aliases are: `CmdToClickButton()`, `CmdToClickBut()`

GetCommandToDumpButton()

```
GetCommandToDumpButton(widgetName, dialog=False,  
level=0)
```

Gets the command that dumps the value of a button defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the `Button` widget.
- `dialog` — If `True`, specifies that the button is defined in a modal dialog box. If `False`, specifies that the button is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

Aliases are: `CmdToDumpButton()`, `CmdToDumpBut()`

IsButtonDimmed()

`IsButtonDimmed(widgetName)`

Determines whether a button in the window is dimmed. This function returns `True` if the button is dimmed or if the button does not exist, and it returns `False` otherwise.

The argument is:

- `widgetName` — Specifies the name of the `Button` widget.

The alias is: `IsBtnDimmed()`

IsButtonDown()

`IsButtonDown(widgetName)`

Determines whether a button in the window is in the “on” state (that is, you have clicked the mouse over the button graphic so that the button appears to be pushed down, or the command to simulate this action has been executed successfully). This function returns `True` if the button is pushed down. It returns `False` if the button is not pushed down or if the button does not exist.

The argument is:

- `widgetName` — Specifies the name of the `Button` widget.

The alias is: `IsBtnDown()`

SelectButton()

`SelectButton(widgetName, block=True, printOutput=True)`

Simulates clicking the specified button and returns `True` on success and `False` otherwise.

Arguments are:

- `widgetName` — Specifies the name of the `Button` widget.

- `block` — If `True`, executes `SelectButton()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

The alias is: `PressButton()`

WaitButtonInStatus()

```
WaitButtonInStatus(widgetName, dimmed, isDown=False,  
duration=-1.0, checkInterval=0.5, verbose=True)
```

Waits until the specified button reaches the specified status. This function returns `True` if the button has reached the specified status, and it returns `False` otherwise.

Arguments are:

- `widgetName` — Specifies the name of the `Button` widget.
- `dimmed` — Indicates that the function waits until the button is dimmed.
- `isDown` — If `True`, indicates that the function waits until the button is in the “on” state (that is, you have clicked the mouse over the button graphic so that the button appears to be pushed down, or the command to simulate this action has been executed successfully). If `False`, indicates that the function waits until the button is in the “off” state (that is, it appears to have popped up).
- `duration` — Specifies how long the function waits, if at all. The `duration` may be:
 - A negative number — Indicates that the function waits until the button reaches the specified status.
 - `0.0` — Indicates that the function does not wait.
 - A positive number — Specifies the maximum number of seconds that the function waits.
- `checkInterval` — Specifies the interval (in seconds) between status checks.

- `verbose` — If `True`, prints detailed error information, if any. If `False`, does not print error information.

GHS_Window Edit and Terminal Widget Functions

The following sections describe the functions from class `GHS_Window` that relate to `Edit` and `Terminal` widgets.



Tip Many of the following functions require you to specify a widget name. You can list widget names with the `ShowWidgets()` function. See “`ShowWidgets()`” on page 149.

GetEditTextLines()

```
GetEditTextLines(widgetName, removeCrs=True)
```

Returns a list of strings for the text in an `Edit` or `Terminal` widget. Each string in the list is a separate line of text. An empty list is returned upon error.

Arguments are:

- `widgetName` — Specifies the name of the `Edit` or `Terminal` widget.
- `removeCrs` — If `True`, removes carriage return characters (if any). If `False`, does not remove carriage return characters.

Aliases are: `EditTextLines()`, `EditLines()`,
`GetTermTextLines()`, `TermTextLines()`, `TermLines()`

GetEditTextString()

```
GetEditTextString(widgetName)
```

Returns a single string for the text in an `Edit` or `Terminal` widget. An empty string (`" "`) is returned upon error.

The argument is:

- `widgetName` — Specifies the name of the `Edit` or `Terminal` widget.

Aliases are: `EditTextString()`, `EditTextStr()`,
`GetTermTextString()`, `TermTextString()`, `TermTextStr()`

GHS_Window MScrollList Widget Functions

The following sections describe the functions from class `GHS_Window` that relate to `MScrollList` widgets.



Tip Many of the following functions require you to specify a widget name. You can list widget names with the `ShowWidgets()` function. See “`ShowWidgets()`” on page 149.

For information about the related functions of class `GHS_MslTree`, see “`GHS_MslTree` Attributes and Functions” on page 141.

ChangeMslTree()

```
ChangeMslTree(widgetName, row, col=-1, expand=True,  
block=True, printOutput=True)
```

Expands or shrinks the sub-tree of an `MScrollList` widget cell. This function returns `True` on success and `False` on failure.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.
- `row` — Specifies the index of the row to expand/collapse. The index starts at 0 (zero). If the row number is negative, the operation is applied to the first row of the current selection.
- `col` — Specifies the index of the column to expand/collapse. The index starts at 0 (zero).
- `expand` — If `True`, expands the sub-tree. If `False`, collapses the sub-tree.
- `block` — If `True`, executes `ChangeMslTree()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

ChangeWholeMslTree()

```
ChangeWholeMslTree(widgetName, col=-1, expand=True,  
recursive=False)
```

Expands or contracts all nodes in the tree of an `MScrollList` widget cell. This function returns `True` on success and `False` on failure.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.
- `col` — This argument has no effect at present.
- `expand` — If `True`, expands the tree. If `False`, contracts the tree.
- `recursive` — If `True`, the expansion/contraction continues until the operation has no effect. If `False`, only expands/contracts the current tree.

DoubleClickMslCell()

```
DoubleClickMslCell(widgetName, row, col, block=True,  
printOutput=True)
```

Simulates double-clicking one or more cells of an `MScrollList` widget defined in the window. This function returns `True` on success and `False` otherwise.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.
- `row` — Specifies the index of the row to double-click. The index starts at 0 (zero). If the row number is less than 0, the first row of the current selection is double-clicked.
- `col` — Specifies the index of the column to double-click. The index starts at 0 (zero).
- `block` — If `True`, executes `DoubleClickMslCell()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

The alias is: `DblClickMslCell()`

DoubleClickMslCellByValue()

```
DoubleClickMslCellByValue(widgetName, cellValue, col,  
block=True, printOutput=True)
```

Double-clicks the `MScrollList` widget cell with the specified value. This function returns `True` on success and `False` on failure.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.
- `cellValue` — Specifies the value of the cell to double-click.
- `col` — Specifies the index of the column to double-click. The index starts at 0 (zero).
- `block` — If `True`, executes `DoubleClickMslCellByValue()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

Aliases are: `DblClickMslCellByValue()`,
`DblClickMslCellByVal()`

DumpMslHighlight()

```
DumpMslHighlight(widgetName, block=True,  
printOutput=True)
```

Dumps the contents of an `MScrollList` widget defined in the window and indicates which cells are highlighted. This function returns `True` on success and `False` otherwise. The dumped contents are kept in the object's `cmdExecOutput` attribute.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.

- `block` — If `True`, executes `DumpMslHighlight()` in blocked mode and grabs the output. If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print output.

The alias is: `DumpMslHl()`

DumpMslSelection()

```
DumpMslSelection(widgetName, block=True,  
printOutput=True)
```

Dumps selected cells of an `MScrollList` widget defined in the window. This function returns `True` on success and `False` otherwise. The dumped contents are kept in the object's `cmdExecOutput` attribute.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.
- `block` — If `True`, executes `DumpMslSelection()` in blocked mode and grabs the output. If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print output.

The alias is: `DumpMslSel()`

DumpMslValue()

```
DumpMslValue(widgetName, block=True,  
printOutput=True)
```

Dumps the contents of an `MScrollList` widget defined in the window. This function returns `True` on success and `False` otherwise. The dumped contents are kept in the object's `cmdExecOutput` attribute.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.

- `block` — If `True`, executes `DumpMslValue()` in blocked mode and grabs the output. If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print output.

The alias is: `DumpMsl()`

ExtendMslSelection()

```
ExtendMslSelection(widgetName, row, col=-1,  
block=True, printOutput=True)
```

Extends the selection in an `MScrollList` widget. This function returns `True` on success and `False` on failure.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.
- `row` — Specifies the index of the row to extend the selection to. The index starts at 0 (zero). If the row number is less than 0, the specified column is selected in all rows.
- `col` — Specifies the index of the column to extend the selection to. The index starts at 0 (zero). If the column number is less than 0, all columns of the specified row are selected.
- `block` — If `True`, executes `ExtendMslSelection()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

If both the row and column numbers are less than 0, all cells of the `MScrollList` widget are selected.



Note This function simulates the behavior of the `Ctrl + left-click` operation, which clears existing selections (if any) on applicable cells.

The alias is: `ExtMslSel()`

GetCommandToChangeMslTree()

```
GetCommandToChangeMslTree(widgetName, row, col,  
expand=True, dialog=False, level=0)
```

Gets the command that expands or collapses the sub-tree associated with an `MScrollList` widget cell. The widget may be defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.
- `row` — Specifies the index of the row to expand/collapse. The index starts at 0 (zero). If the row number is negative, the operation is applied to the current selection.
- `col` — Specifies the index of the column to expand/collapse. The index starts at 0 (zero).
- `expand` — If `True`, specifies to expand the sub-tree. If `False`, specifies to collapse the sub-tree.
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

The alias is: `CmdToChangeMslTree()`

GetCommandToDoubleClickMslCell()

```
GetCommandToDoubleClickMslCell(widgetName, row, col,  
dialog=False, level=0)
```

Gets the command that double-clicks one or more cells of an `MScrollList` widget defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.
- `row` — Specifies the index of the row to double-click. The index starts at 0 (zero).
- `col` — Specifies the index of the column to double-click. The index starts at 0 (zero).
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

The alias is: `CmdToDbClickMslCell()`

GetCommandToDoubleClickMslCellByValue()

`GetCommandToDoubleClickMslCellByValue(widgetName, cellValue, col, dialog=False, level=0)`

Gets the command that double-clicks the `MScrollList` widget cell with the specified value. The widget may be defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between `MULTI` releases.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.
- `cellValue` — Specifies the value of the cell to double-click.
- `col` — Specifies the index of the column to double-click. The index starts at 0 (zero).
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

Aliases are: `CmdToDbClickMslCellByValue()`,
`CmdToDbClickMslCellByVal()`

GetCommandToDumpMsl()

```
GetCommandToDumpMsl (widgetName, dialog=False,  
level=0)
```

Gets the command that dumps the contents of an `MScrollList` widget defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

The alias is: `CmdToDumpMsl ()`

GetCommandToDumpMslHighlight()

```
GetCommandToDumpMslHighlight (widgetName,  
dialog=False, level=0)
```

Gets the command that dumps the contents of an `MScrollList` widget defined in the window or in a modal dialog box and that indicates which cells are highlighted. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

The alias is: `CmdToDumpMslHl ()`

GetCommandToDumpMslSelection()

```
GetCommandToDumpMslSelection(widgetName,  
dialog=False, level=0)
```

Gets the command that dumps selected cells of an `MScrollList` widget defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

The alias is: `CmdToDumpMslSel()`

GetCommandToExtendMslSelection()

```
GetCommandToExtendMslSelection(widgetName, row,  
col=-1, dialog=False, level=0)
```

Gets the command that extends the selection in an `MScrollList` widget. The widget may be defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.
- `row` — Specifies the index of the row to extend the selection to. The index starts at 0 (zero). If the row number is less than 0, the specified column is selected in all rows.
- `col` — Specifies the index of the column to extend the selection to. The index starts at 0 (zero). If the column number is less than 0, all columns of the specified row are selected.

- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

If both the row and column numbers are less than 0, all cells of the `MScrollList` widget are selected.



Note The command simulates the behavior of the `Ctrl+left-click` operation, which clears existing selections (if any) on applicable cells.

The alias is: `CmdToExtMslSel()`

GetCommandToSelectMslCell()

```
GetCommandToSelectMslCell(widgetName, row, col=-1,  
dialog=False, level=0)
```

Gets the command that selects cells of an `MScrollList` widget defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between `MULTI` releases.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.
- `row` — Specifies the index of the row to select. The index starts at 0 (zero). If the row number is less than 0 and the widget supports single-cell selection and multiple-row selection, the specified column is selected in all rows.
- `col` — Specifies the index of the column to select. The index starts at 0 (zero). If the column number is less than 0, all columns of the specified row are selected.
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

The alias is: `CmdToSelMslCell()`

GetCommandToSelectMslCellByValue()

```
GetCommandToSelectMslCellByValue(widgetName,  
cellValue, col=0, dialog=False, level=0)
```

Gets the command that selects an `MScrollList` widget cell by value. The widget may be defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.
- `cellValue` — Specifies the value of the cell to select.
- `col` — Specifies the index of the column whose cell values are searched for `cellValue`. The index starts at 0 (zero). The value specified should be a valid column index. A negative value results in selection failure.
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

Aliases are: `CmdToSelMslCellByValue()`,
`CmdToSelMslCellByVal()`

GetCommandToSortMsl()

```
GetCommandToSortMsl(widgetName, col, dialog=False,  
level=0)
```

Gets the command that sorts an `MScrollList` widget defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.

- `col` — Specifies the index of the column to sort. The index starts at 0 (zero). The value specified should be a valid column index; a negative value is invalid.
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

The alias is: `CmdToSortMsl()`

GetMslRowNumber()

`GetMslRowNumber(widgetName)`

Gets the number of rows in an `MScrollList` widget.

The argument is:

- `widgetName` — Specifies the name of the `MScrollList` widget.

The alias is: `GetMslRowNum()`

GetMslTree()

`GetMslTree(widgetName, treeNodeColumn=0)`

Dumps the content of an `MScrollList` widget and parses it into a `GHS_MslTree` object. For information about `GHS_MslTree`, see Chapter 9, “Widget Functions”.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.
- `treeNodeColumn` — Specifies the column on which the tree in the `MScrollList` widget is built. Usually, the tree in an `MScrollList` widget is built on column 0 (zero).

SelectMslCell()

```
SelectMslCell(widgetName, row, col=-1, block=True,
printOutput=True)
```

Selects cells of an `MScrollList` widget defined in the window. This function returns `True` on success and `False` otherwise.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.
- `row` — Specifies the index of the row to select. The index starts at 0 (zero). If the row number is less than 0 and the widget supports single-cell selection and multiple-row selection, the specified column is selected in all rows.
- `col` — Specifies the index of the column to select. The index starts at 0 (zero). If the column number is less than 0, all columns of the specified row are selected.
- `block` — If `True`, executes `SelectMslCell()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

The alias is: `SelMslCell()`

SelectMslCellByValue()

```
SelectMslCellByValue(widgetName, cellValue, col=0,
block=True, printOutput=True)
```

Selects an `MScrollList` widget cell by value. The widget should be defined in the window. This function returns `True` on success and `False` otherwise.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.
- `cellValue` — Specifies the value of the cell to select.
- `col` — Specifies the index of the column whose cell values are searched for `cellValue`. The index starts at 0 (zero). The value specified should be a valid column index. A negative value results in selection failure.

- `block` — If `True`, executes `SelectMslCellByValue()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

The alias is: `SelMslCellByValue()`

SortMslByColumn()

```
SortMslByColumn(widgetName, col, block=True,  
printOutput=True)
```

Sorts an `MScrollList` widget by a column. The widget should be defined in the window. This function returns `True` on success and `False` otherwise.

Arguments are:

- `widgetName` — Specifies the name of the `MScrollList` widget.
- `col` — Specifies the index of the column whose values are sorted. The index starts at 0 (zero). The value specified should be a valid column index. A negative value is invalid.
- `block` — If `True`, executes `SortMslByColumn()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

Aliases are: `SortMslByCol()`, `SortMsl()`

GHS_Window PullDown Widget Functions

The following sections describe the functions from class `GHS_Window` that relate to `PullDown` widgets.



Tip Many of the following functions require you to specify a widget name. You can list widget names with the `ShowWidgets()` function. See “`ShowWidgets()`” on page 149.

ChangePullDownValue()

```
ChangePullDownValue(widgetName, value)
```

Changes the value of a `PullDown` widget defined in the window and returns `True` on success and `False` otherwise.

Arguments are:

- `widgetName` — Specifies the name of the `PullDown` widget.
- `value` — Specifies the widget’s new value.

Aliases are: `ChangePdValue()`, `ChangePdVal()`

DumpPullDownMenu()

```
DumpPullDownMenu(widgetName, block=True,  
printOutput=True)
```

Dumps the name, menu, etc. of a `PullDown` widget defined in the window and returns `True` on success and `False` otherwise.

Arguments are:

- `widgetName` — Specifies the name of the `PullDown` widget.
- `block` — If `True`, executes `DumpPullDownMenu()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

The alias is: `DumpPdMenu()`

DumpPullDownValue()

```
DumpPullDownValue(widgetName, block=True,  
printOutput=True)
```

Dumps the name, value, etc. of a PullDown widget defined in the window and returns True on success and False otherwise.

Arguments are:

- `widgetName` — Specifies the name of the PullDown widget.
- `block` — If True, executes `DumpPullDownValue()` in blocked mode and grabs the output (if any). If False, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If True, prints the output. If False, does not print the output.

The alias is: `DumpPdValue()`

GetCommandToChangePullDownValue()

```
GetCommandToChangePullDownValue(widgetName, value,  
dialog=False, level=0)
```

Gets the command that changes the value of a PullDown widget defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the PullDown widget.
- `value` — Specifies the new value of the widget.
- `dialog` — If True, specifies that the widget is defined in a modal dialog box. If False, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

Aliases are: `CmdToChangePdValue()`, `CmdToChangePdVal()`

GetCommandToDumpPullDownMenu()

```
GetCommandToDumpPullDownMenu(widgetName,  
dialog=False, level=0)
```

Gets the command that dumps a PullDown widget's menu. The widget may be defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the PullDown widget.
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

Aliases are: `CmdToDumpPdMenu()`

GetCommandToDumpPullDownValue()

```
GetCommandToDumpPullDownValue(widgetName,  
dialog=False, level=0)
```

Gets the command that dumps the value of a PullDown widget defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the PullDown widget.
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

Aliases are: `CmdToDumpPdValue()`, `CmdToDumpPdVal()`

GetCommandToSelectPullDownMenu()

```
GetCommandToSelectPullDownMenu(widgetName, valIdx,  
dialog=False, level=0)
```

Gets the command that selects a PullDown widget's menu item. The widget may be defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the PullDown widget.
- `valIdx` — Specifies the index of the menu item. The index of the first menu item is 0 (zero).
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

The alias is: `CmdToSelPdMenu()`

GetPullDownMenu()

```
GetPullDownMenu(widgetName)
```

Gets the menu entries of a PullDown widget defined in the window. This function returns the widget's menu entries as a list, or it returns an empty list upon error.

The argument is:

- `widgetName` — Specifies the name of the PullDown widget.

The alias is: `GetPdMenu()`

GetPullDownValue()

`GetPullDownValue(widgetName)`

Gets the value of a `PullDown` widget defined in the window. This function returns the widget's value as a string, or it returns an empty string ("") upon error.

The argument is:

- `widgetName` — Specifies the name of the `PullDown` widget.

Aliases are: `GetPdValue()`, `GetPdVal()`

SelectPullDownValue()

`SelectPullDownValue(widgetName, valIdx)`

Selects a `PullDown` widget's value from the value list (menu) in the window. This function returns `True` on success and `False` otherwise.

Arguments are:

- `widgetName` — Specifies the name of the `PullDown` widget.
- `valIdx` — Specifies the index of the value.

Aliases are: `SelPdValue()`, `SelPdVal()`

GHS_Window Tab Widget Functions

The following sections describe the functions from class `GHS_Window` that relate to Tab widgets.



Tip Many of the following functions require you to specify a widget name. You can list widget names with the `ShowWidgets()` function. See “`ShowWidgets()`” on page 149.

DumpTabContents()

```
DumpTabContents(widgetName="", block=True,  
printOutput=True)
```

Dumps the current tab contents of a Tab widget defined in the window. This function returns `True` on success and `False` otherwise.

Arguments are:

- `widgetName` — Specifies the name of the Tab widget.
- `block` — If `True`, executes `DumpTabContents()` in blocked mode and grabs the output, if any. If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

The alias is: `DumpTab()`

DumpTabSelection()

```
DumpTabSelection(widgetName="", block=True,  
printOutput=True)
```

Dumps the current tab name of a Tab widget defined in the window. As shown in the following example output, the tab name is preceded by a short description:

```
Py Out: Selected tab name: Debugger
```

If you only want the tab name (and not the description), use `GetTabSelection()` instead. See “`GetTabSelection()`” on page 179.

This function returns `True` on success and `False` otherwise.

Arguments are:

- `widgetName` — Specifies the name of the Tab widget.
- `block` — If `True`, executes `DumpTabSelection()` in blocked mode and grabs the output, if any. If `False`, neither executes the function in blocked mode nor grabs the output.

- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

The alias is: `DumpTabSel()`

DumpTabValue()

```
DumpTabValue(widgetName="", block=True,
printOutput=True)
```

Dumps the tab names of a Tab widget defined in the window. This function returns a string for the tab names. As shown in the following example output, the tab names are preceded by a short description:

```
Py Out: Tabs: General, Project Manager, Debugger
```

If you only want the tab names (and not the description), use `GetTabNames()` instead. See “`GetTabNames()`” on page 179.

Arguments are:

- `widgetName` — Specifies the name of the Tab widget.
- `block` — If `True`, executes `DumpTabValue()` in blocked mode and grabs the output. If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

The alias is: `DumpTabVal()`

GetCommandToDumpTab()

```
GetCommandToDumpTab(widgetName, dialog=False,
level=0)
```

Gets the command that dumps the contents of a Tab widget’s current page. The widget may be defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the Tab widget.
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

The alias is: `CmdToDumpTab()`

GetCommandToDumpTabSelection()

```
GetCommandToDumpTabSelection(widgetName,  
dialog=False, level=0)
```

Gets the command that dumps a descriptive string and the current tab name of a Tab widget. The widget may be defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases. For more information about the descriptive string, see “DumpTabSelection()” on page 175.

Arguments are:

- `widgetName` — Specifies the name of the Tab widget.
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

The alias is: `CmdToDumpTabSel()`

GetCommandToDumpTabValue()

```
GetCommandToDumpTabValue(widgetName, dialog=False,  
level=0)
```

Gets the command that dumps the tab names of a Tab widget that is defined in the window or in a modal dialog box. This function returns a string for the

command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the Tab widget.
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

The alias is: `CmdToDumpTabVal()`

GetCommandToSelectTab()

```
GetCommandToSelectTab(widgetName, tabName,  
dialog=False, level=0)
```

Gets the command that selects a tab of a Tab widget defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the Tab widget.
- `tabName` — Specifies the name of the tab (page) to select.
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

The alias is: `CmdToSelTab()`

GetTabNames()

`GetTabNames(widgetName)`

Gets the tab names of a Tab widget defined in the window. This function returns a list of tab names or an empty list upon error.

The argument is:

- `widgetName` — Specifies the name of the Tab widget.

See also “`DumpTabValue()`” on page 176.

GetTabSelection()

`GetTabSelection(widgetName)`

Gets the name for the current tab of a Tab widget defined in the window. This function returns a string for the current tab or an empty string (" ") upon error.

The argument is:

- `widgetName` — Specifies the name of the Tab widget.

See also “`DumpTabSelection()`” on page 175.

Aliases are: `GetTabSel()`, `GetCurrentTab()`, `GetCurTab()`

SelectTab()

`SelectTab(widgetName, tabName, block=True, printOutput=True)`

Selects the tab of a Tab widget by name. The widget should be defined in the window. This function returns `True` on success and `False` otherwise.

Arguments are:

- `widgetName` — Specifies the name of the Tab widget.
- `tabName` — Specifies the name of the tab (page) to be selected.

- `block` — If `True`, executes `SelectTab()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

The alias is: `SelTab()`

GHS_Window Text Widget Functions

The following sections describe the functions from class `GHS_Window` that relate to `Text` widgets.



Tip The following functions require you to specify a widget name. You can list widget names with the `ShowWidgets()` function. See “`ShowWidgets()`” on page 149.

GetCommandToDumpText()

```
GetCommandToDumpText(widgetName, dialog=False,
level=0)
```

Gets the command that dumps the value of a `Text` widget defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between `MULTI` releases.

Arguments are:

- `widgetName` — Specifies the name of the `Text` widget.
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

See also “`GetTextValue()`” on page 181.

The alias is: `CmdToDumpTx()`

GetTextValue()

`GetTextValue(widgetName)`

Gets the value of a `Text` widget defined in the window. This function returns the widget's value as a string, or it returns an empty string ("") upon error.

The argument is:

- `widgetName` — Specifies the name of the `Text` widget.

See also “`GetCommandToDumpText()`” on page 180.

The alias is: `GetTxVal()`

GHS_Window TextCell Widget Functions

The following sections describe the functions from class `GHS_Window` that relate to `TextCell` widgets.



Tip Many of the following functions require you to specify a widget name. You can list widget names with the `ShowWidgets()` function. See “`ShowWidgets()`” on page 149.

DumpTextCellValue()

`DumpTextCellValue(widgetName, block=True, printOutput=True)`

Dumps the value, widget name, etc. of a `TextCell` widget defined in the window. This function returns `True` on success and `False` otherwise.

Arguments are:

- `widgetName` — Specifies the name of the `TextCell` widget.
- `block` — If `True`, executes `DumpTextCellValue()` in blocked mode and grabs the output, if any. If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

See also “GetTextCellValue()” on page 182.

Aliases are: `DumpTcValue()`, `DumpTcVal()`

GetTextCellValue()

`GetTextCellValue(widgetName)`

Gets the value of a `TextCell` widget defined in the window. This function returns the widget’s value as a string, or it returns an empty string (“”) upon error.

The argument is:

- `widgetName` — Specifies the name of the `TextCell` widget.

See also “DumpTextCellValue()” on page 181.

Aliases are: `GetTcValue()`, `GetTcVal()`

IsTextCellReadOnly()

`IsTextCellReadOnly(widgetName)`

Checks if a `TextCell` widget defined in the window or in a modal dialog box is read-only. This function returns `True` if yes or if the given widget is not a `TextCell`, and `False` otherwise.

The argument is:

- `widgetName` — Specifies the name of the `TextCell` widget.

The alias is: `IsTcReadOnly()`

GHS_Window TextField Widget Functions

The following sections describe the functions from class `GHS_Window` that relate to `TextField` widgets.



Tip Many of the following functions require you to specify a widget name. You can list widget names with the `ShowWidgets()` function. See “`ShowWidgets()`” on page 149.

ChangeTextFieldValue()

```
ChangeTextFieldValue(widgetName, value,  
hitReturn=True, block=True, printOutput=True)
```

Changes the value of a `TextField` widget defined in the window or in a modal dialog box. This function returns `True` on success and `False` otherwise.

Arguments are:

- `widgetName` — Specifies the name of the `TextField` widget.
- `value` — Specifies the new value of the widget.
- `hitReturn` — If `True`, simulates pressing `Enter` on the `TextField` widget. If `False`, does not simulate pressing `Enter`.
- `block` — If `True`, executes `ChangeTextFieldValue()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

Aliases are: `ChangeTfValue()`, `ChangeTfVal()`

DumpTextFieldValue()

```
DumpTextFieldValue(widgetName, block=True,  
printOutput=True)
```

Dumps the value, widget name, etc. of a `TextField` widget defined in the window. This function returns `True` on success and `False` otherwise.

Arguments are:

- `widgetName` — Specifies the name of the `TextField` widget.
- `block` — If `True`, executes `DumpTextFieldValue()` in blocked mode and grabs the output, if any. If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

See also “`GetTextFieldValue()`” on page 186.

Aliases are: `DumpTfValue()`, `DumpTfVal()`

GetCommandToChangeTextFieldValue()

```
GetCommandToChangeTextFieldValue(widgetName, value,  
hitReturn=True, dialog=False, level=0)
```

Gets the command that changes the value of a `TextField` widget defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between `MULTI` releases.

Arguments are:

- `widgetName` — Specifies the name of the `TextField` widget.
- `value` — Specifies the widget’s new value.
- `hitReturn` — If `True`, simulates pressing Enter on the `TextField` widget. If `False`, does not simulate pressing Enter.
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

Aliases are: `CmdToChangeTfValue()`, `CmdToChangeTfVal()`

GetCommandToDumpTextField()

```
GetCommandToDumpTextField(widgetName, dialog=False,  
level=0)
```

Gets the command that dumps the value of a `TextField` widget defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the `TextField` widget.
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

The alias is: `CmdToDumpTf()`

GetCommandToReturnOnTextField()

```
GetCommandToReturnOnTextField(widgetName,  
dialog=False, level=0)
```

Gets the command that simulates pressing Enter on a `TextField` widget defined in the window or in a modal dialog box. This function returns a string for the command. The command that is returned is not documented and should not be relied upon between MULTI releases.

Arguments are:

- `widgetName` — Specifies the name of the `TextField` widget.
- `dialog` — If `True`, specifies that the widget is defined in a modal dialog box. If `False`, specifies that the widget is defined in the window.
- `level` — Specifies the nested level of the modal dialog box. For more information, see the note at the beginning of this chapter (page 139).

Aliases are: `CmdToReturnOnTf()`, `CmdToRetOnTf()`

GetTextFieldValue()

`GetTextFieldValue(widgetName)`

Gets the value of a `TextField` widget defined in the window. This function returns the widget's value as a string, or it returns an empty string ("") upon error.

Arguments are:

- `widgetName` — Specifies the name of the `TextField` widget.

See also “`DumpTextFieldValue()`” on page 183.

Aliases are: `GetTfValue()`, `GetTfVal()`

IsTextFieldReadOnly()

`IsTextFieldReadOnly(widgetName)`

Checks if a `TextField` widget defined in the window or in a modal dialog box is read-only. This function returns `True` if yes or if the given widget is not a `TextField`, and `False` otherwise.

The argument is:

- `widgetName` — Specifies the name of the `TextField` widget.

The alias is: `IsTfReadOnly()`

ReturnOnTextField()

`ReturnOnTextField(widgetName, block=True, printOutput=True)`

Simulates pressing Enter on a `TextField` widget defined in the window. This function returns `True` on success and `False` otherwise.

Arguments are:

- `widgetName` — Specifies the name of the `TextField` widget.

- `block` — If `True`, executes `ReturnOnTextField()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

Aliases are: `ReturnOnTf()`, `RetOnTf()`

Window Tracking Functions

Chapter 10

This Chapter Contains:

- GHS_WindowRegister Basic Functions
- GHS_WindowRegister Check Functions
- GHS_WindowRegister Get Window Functions
- GHS_WindowRegister Interactive Functions
- GHS_WindowRegister Window Manipulation Functions
- GHS_WindowRegister Wait Functions

This chapter documents functions from class `GHS_WindowRegister`. Class `GHS_WindowRegister` inherits from class `GHS_IdeObject` and implements the mechanism for tracking MULTI IDE windows. All operations applied to MULTI IDE windows are dispatched to the corresponding MULTI IDE components via this mechanism.

The `GHS_WindowRegister` functions are divided into the following sections:

- “`GHS_WindowRegister` Basic Functions” on page 190
- “`GHS_WindowRegister` Check Functions” on page 191
- “`GHS_WindowRegister` Get Window Functions” on page 193
- “`GHS_WindowRegister` Interactive Functions” on page 201
- “`GHS_WindowRegister` Window Manipulation Functions” on page 206
- “`GHS_WindowRegister` Wait Functions” on page 207

`GHS_WindowRegister` Basic Functions

The following section describes the function from class `GHS_WindowRegister`.

`__init__()`

```
__init__(workingDir="")
```

Initializes object attributes.

The argument is:

- `workingDir` — Stores the working directory of the MULTI IDE service. It is not used at present.

GHS_WindowRegister Check Functions

The following sections describe the functions from class `GHS_WindowRegister` that check whether the specified window exists in the system or exists in the given window list.

CheckWindow()

```
CheckWindow(winName, winId=0, winClass="",  
winRegSvcName="", pid=0, fromWinList=None,  
notInWinList=None)
```

Checks that a MULTI IDE window is in one window list and not in another. If this is the case, this function returns a `GHS_Window` object; otherwise, it returns `None`.

Arguments are:

- `winName` — Specifies the name of the window. An empty string ("") matches any window name.
- `winId` — Specifies the internal ID of the window. A 0 (zero) matches any window ID.
- `winClass` — Specifies the class of the window. An empty string ("") matches any window class.
- `winRegSvcName` — Specifies the internal ID of the component to which the window belongs. An empty string ("") matches any internal ID.
- `pid` — Specifies the PID of the process to which the window belongs. A 0 (zero) matches any PID.
- `fromWinList` — Specifies the window list that should contain the given window. If an empty window list is given, the window list for the current system is used.
- `notInWinList` — Specifies the window list that should not contain the given window.

The alias is: `CheckWin()`

CheckWindowObject()

`CheckWindowObject (winObj)`

Checks whether a specified GHS_Window object exists in the system. If yes, this function returns the GHS_Window object; otherwise, it returns None.

The argument is:

- `winObj` — Specifies the GHS_Window object to be checked.

Aliases are: `CheckWindowObj()`, `CheckWinObj()`

IsWindowInList()

`IsWindowInList (winList, winName, winId=0, winClass="", pid=0, winRegSvcName="")`

Checks whether the specified window exists in a window list. This function returns True if yes, and False otherwise.

Arguments are:

- `winList` — Specifies the window list to check against.
- `winName` — Specifies the name of the window. An empty string ("") matches any window name.
- `winId` — Specifies the internal ID of the window. A 0 (zero) matches any window ID.
- `winClass` — Specifies the class of the window. An empty string ("") matches any window class.
- `pid` — Specifies the PID of the process to which the window belongs. A 0 (zero) matches any PID.
- `winRegSvcName` — Specifies the internal ID of the component to which the window belongs. An empty string ("") matches any internal ID.

GHS_WindowRegister Get Window Functions

The following sections describe the functions from class `GHS_WindowRegister` that get windows and return window objects. (A couple of the functions listed relate to getting and displaying window lists rather than windows.)

The following arguments are common to these functions:

- `winName` — Specifies a regular expression matching the name of the window. For example, `"^My Window$"` matches “My Window” exactly if there are multiple window names containing the string “My Window”. An empty string (`" "`) matches any window name.
- `pid` — Specifies the PID of the process to which the window belongs. A 0 (zero) matches any PID.
- `fromWinList` — Specifies the window list that contains the given window. If this argument is `None` or empty, the window list for the current system is used.
- `notInWinList` — Specifies the window list that does not contain the given window.
- `warnIfNotFound` — If `True`, prints a warning message when the desired window is not found. If `False`, does not print a warning message.

GetCheckoutBrowserWindow()

```
GetCheckoutBrowserWindow(winName="", pid=0,  
fromWinList=None, notInWinList=None,  
warnIfNotFound=True)
```

Gets a MULTI Checkout Browser window from the given window list and returns the created `GHS_CoBrowseWindow` object.

For argument descriptions, see “GHS_WindowRegister Get Window Functions” on page 193.

Aliases are: `GetCheckoutBrowser()`, `GetCoB()`

GetConnectionOrganizerWindow()

```
GetConnectionOrganizerWindow(winName="",  
pid=0, fromWinList=None, notInWinList=None,  
warnIfNotFound=True)
```

Gets a MULTI Connection Organizer window from the given window list and returns the created `GHS_ConnectionOrganizerWindow` object.

For argument descriptions, see “GHS_WindowRegister Get Window Functions” on page 193.

Aliases are: `GetConnectionOrganizer()`, `GetCo()`

GetDebuggerWindow()

```
GetDebuggerWindow(winName="", pid=0,  
fromWinList=None, notInWinList=None,  
warnIfNotFound=True)
```

Gets a MULTI Debugger window from the given window list and returns the created `GHS_DebuggerWindow` object.

For argument descriptions, see “GHS_WindowRegister Get Window Functions” on page 193.

Aliases are: `GetDebuggerWin()`, `GetDebugger()`

GetDialogByName()

```
GetDialogByName(winName="", pid=0, fromWinList=None,  
notInWinList=None, warnIfNotFound=True)
```

Gets a MULTI IDE dialog box by its name and returns the created `GHS_Window` object.

For argument descriptions, see “GHS_WindowRegister Get Window Functions” on page 193.

Aliases are: `GetDlgByName()`, `GetDialog()`, `GetDlg()`

GetDiffViewerWindow()

```
GetDiffViewerWindow(winName="", pid=0,  
fromWinList=None, notInWinList=None,  
warnIfNotFound=True)
```

Gets a MULTI Diff Viewer window from the given window list and returns the created `GHS_DiffViewWindow` object.

For argument descriptions, see “GHS_WindowRegister Get Window Functions” on page 193.

Aliases are: `GetDiffViewer()`, `GetDv()`

GetEditorWindow()

```
GetEditorWindow(winName="", pid=0, fromWinList=None,  
notInWinList=None, warnIfNotFound=True)
```

Gets a MULTI Editor window from the given window list and returns the created `GHS_EditorWindow` object.

For argument descriptions, see “GHS_WindowRegister Get Window Functions” on page 193.

Aliases are: `GetEditorWin()`, `GetEditor()`

GetEventAnalyzerWindow()

```
GetEventAnalyzerWindow(winName="", pid=0,  
fromWinList=None, notInWinList=None,  
warnIfNotFound=True)
```

Gets a MULTI EventAnalyzer window from the given window list and returns the created `GHS_EventAnalyzerWindow` object.

For argument descriptions, see “GHS_WindowRegister Get Window Functions” on page 193.

Aliases are: `GetEventAnalyzerWin()`, `GetMeaWindow()`,
`GetMeaWin()`, `GetMea()`, `GetMev()`

GetHelpViewerWindow()

```
GetHelpViewerWindow(winName="", pid=0,  
fromWinList=None, notInWinList=None,  
warnIfNotFound=True)
```

Gets a MULTI Help Viewer window from the given window list and returns the created GHS_HelpViewerWindow object.

For argument descriptions, see “GHS_WindowRegister Get Window Functions” on page 193.

Aliases are: `GetHelpViewer()`, `GetHv()`

GetLauncherWindow()

```
GetLauncherWindow(winName="", pid=0,  
fromWinList=None, notInWinList=None,  
warnIfNotFound=True)
```

Gets a MULTI Launcher window from the given window list and returns the created GHS_LauncherWindow object.

For argument descriptions, see “GHS_WindowRegister Get Window Functions” on page 193.

Aliases are: `GetLauncherWin()`, `GetLauncher()`

GetOsaExplorerWindow()

```
GetOsaExplorerWindow(winName="", pid=0,  
fromWinList=None, notInWinList=None,  
warnIfNotFound=True)
```

Gets a MULTI OSA Explorer window from the given window list and returns the created GHS_OsaWindow object.

For argument descriptions, see “GHS_WindowRegister Get Window Functions” on page 193.

Aliases are: `GetOsaExplorer()`, `GetOsa()`

GetProjectManagerWindow()

```
GetProjectManagerWindow(winName="", pid=0,  
fromWinList=None, notInWinList=None,  
warnIfNotFound=True)
```

Gets a MULTI Project Manager window from the given window list and returns the created `GHS_ProjectManagerWindow` object.

For argument descriptions, see “GHS_WindowRegister Get Window Functions” on page 193.

Aliases are: `GetProjMgrWin()`, `GetBuilderWindow()`,
`GetBuilderWin()`, `GetBuilder()`

GetPythonGuiWindow()

```
GetPythonGuiWindow(winName="", pid=0,  
fromWinList=None, notInWinList=None,  
warnIfNotFound=True)
```

Gets a Python GUI window from the given window list and returns the created `GHS_PyGuiWindow` object.

For argument descriptions, see “GHS_WindowRegister Get Window Functions” on page 193.

The alias is: `GetPyGui()`

GetResourceAnalyzerWindow()

```
GetResourceAnalyzerWindow(winName="",  
pid=0, fromWinList=None, notInWinList=None,  
warnIfNotFound=True)
```

Gets a MULTI ResourceAnalyzer window from the given window list and returns the created `GHS_ResourceAnalyzerWindow` object.

For argument descriptions, see “GHS_WindowRegister Get Window Functions” on page 193.

Aliases are: `GetResourceAnalyzer()`, `GetMra()`, `GetMrv()`

GetTaskManagerWindow()

```
GetTaskManagerWindow(winName="", pid=0,  
fromWinList=None, notInWinList=None,  
warnIfNotFound=True)
```

Gets a MULTI Task Manager window from the given window list and returns the created GHS_TaskManagerWindow object.

For argument descriptions, see “GHS_WindowRegister Get Window Functions” on page 193.

Aliases are: `GetTaskManagerWin()`, `GetTm()`

GetTerminalWindow()

```
GetTerminalWindow(winName="", pid=0,  
fromWinList=None, notInWinList=None,  
warnIfNotFound=True)
```

Gets an MTerminal window from the given window list and returns the created GHS_TerminalWindow object.

For argument descriptions, see “GHS_WindowRegister Get Window Functions” on page 193.

Aliases are: `GetTerminal()`, `GetTerm()`

GetTraceWindow()

```
GetTraceWindow(winName="", pid=0, fromWinList=None,  
notInWinList=None, warnIfNotFound=True)
```

Gets a MULTI Trace List window from the given window list and returns the created GHS_TraceWindow object.

For argument descriptions, see “GHS_WindowRegister Get Window Functions” on page 193.

The alias is: `GetTrace()`

GetWindowByIndex()

```
GetWindowByIndex(idx, winList=None)
```

Gets a GHS_Window object from an entry in a window list and returns the created GHS_Window object.

Arguments are:

- `idx` — Specifies the index of the entry in the window list.
- `winList` — Specifies the window list. If an empty window list is given, the window list in the current system is used.

Aliases are: `GetWinByIdx()`, `WindowForIndex()`, `WinForIdx()`, `WinFIdx()`

GetWindowByName()

```
GetWindowByName(winName="", pid=0, fromWinList=None,  
notInWinList=None, warnIfNotFound=True)
```

Gets a MULTI IDE window by its name and returns the created GHS_Window object.

For argument descriptions, see “GHS_WindowRegister Get Window Functions” on page 193.

Aliases are: `GetWinByName()`, `GetWindow()`, `GetWin()`

GetWindowList()

```
GetWindowList(showIt=False, showPid=True,  
showClassName=True, showWinId=False,  
showServiceName=False, showParentWin=False,  
showGuiClass=False)
```

Gets the window list for the current MULTI IDE session and returns a window list object.

Arguments are:

- `showIt` — If `True`, displays the window list. The remaining arguments control what items are displayed. If `False`, does not display the window list.
- `showPid` — If `True`, displays the windows' corresponding process IDs (PIDs). If `False`, does not display the PIDs.
- `showClassName` — If `True`, displays the windows' class names. If `False`, does not display the class names.
- `showWinId` — If `True`, displays the windows' internal IDs. If `False`, does not display internal IDs
- `showServiceName` — If `True`, displays the internal IDs of the components to which the windows belong. If `False`, does not display internal IDs.
- `showParentWin` — If `True`, displays the windows' parent window IDs. If `False`, does not display parent window IDs.
- `showGuiClass` — If `True`, displays windows' GUI class names. If `False`, does not display GUI class names.

MULTI windows have two class names: one for their actual category, and another for the category they are shown as in the GUI (such as in menus). The two class names are usually the same, but they differ for some windows. For example, the MULTI Project Manager's progress window is a normal window but should be shown in menus as being part of the Project Manager category.

Aliases are: `GetWinList()`, `GetWindows()`, `GetWins()`, `Windows()`, `Wins()`

ShowWindowList()

```
ShowWindowList(showPid=True, showClassName=True,  
showWinId=False, showServiceName=False,  
showParentWin=False, showGuiClass=False)
```

Displays the window list for the current MULTI session.

For argument descriptions, see “GetWindowList()” on page 199.

Aliases are: `ShowWindows()`, `ShowWins()`

GHS_WindowRegister Interactive Functions

The following sections describe the interactive functions from class `GHS_WindowRegister`.

Beep()

`Beep(count=1, block=False, hostWin=None)`

Beeps the specified number of times and returns `True` on success and `False` on failure.

Arguments are:

- `count` — Specifies the number of beeps.
- `block` — If `True`, executes `Beep()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `hostWin` — the host window from which the dialog is to be shown. For more information, see “`ShowMessage()`” on page 205.

ChooseDir()

`ChooseDir(dftDir="", prompt="Choose directory:", title="Directory Chooser", hostWin=None)`

Allows you to choose a directory via MULTI’s directory chooser. This function returns the selected directory, or it returns an empty string (“”) upon failure or cancellation.

Arguments are:

- `dftDir` — Specifies the default directory.
- `prompt` — Specifies the prompt string to display.
- `title` — Specifies the title of the modal directory chooser. If `title` is an empty string (“”), the value of `prompt` is used as the title.

- `hostWin` — Specifies the host window from which the directory chooser is shown. If the argument is `None` or if it is not a `GHS_Window` object, a window is randomly selected as the host window.

The alias is: `DirChooser()`

ChooseFile()

```
ChooseFile(dftFile="", dftDir="", label="OK",  
forOpen=True, existingFile=False, extension="",  
fileTypes="", eraseFilenameWhenDirChange=False,  
title="File Chooser", hostWin=None)
```

Allows you to choose a file path via MULTI's file chooser. This function returns the selected file path, or it returns an empty string ("") upon failure or cancellation.

Arguments are:

- `dftFile` — Specifies the default filename.
- `dftDir` — Specifies the default directory.
- `label` (UNIX only) — Specifies the label of the action button in the file chooser.
- `forOpen` — Indicates whether the selected file can be read.
- `existingFile` — If `True`, the selected file must already exist. If `False`, the user may create a new file.
- `extension` — Specifies the extension for the selected file.
- `fileTypes` — Specifies the file's MULTI IDE file type.
- `eraseFilenameWhenDirChange` — If `True`, erases the filename located in the file chooser when the directory changes. If `False`, does not erase the filename when the directory changes.
- `title` — Specifies the title of the modal file chooser. If `title` is an empty string (""), `File Chooser` is used as the title.
- `hostWin` — Specifies the host window from which the file chooser is shown. If the argument is `None` or if it is not a `GHS_Window` object, a window is randomly selected as the host window.

The alias is: `FileChooser()`

ChooseFromList()

```
ChooseFromList(dftValueIdx=0, valList=[],  
colValueSep="", colNames=[], prompt="Select value  
from the list:", title="Choose Value from List",  
helpkey="", hostWin=None)
```

Allows you to choose a value from a list displayed in a modal dialog box. This function returns the string selected from the list, or it returns an empty string (" ") upon failure or cancellation. See also “ChooseFromList()” on page 110.

Arguments are:

- `dftValueIdx` — Specifies the index of the list’s default value.
- `valList` — Specifies a list of pre-defined values.
- `colValueSep` — Specifies a column-value separator. If `colValueSep` is an empty string (" "), # is used as the separator by default.
- `colNames` — Specifies the column names. This should be a list of strings.
- `prompt` — Specifies the prompt string to display.
- `title` — Specifies the title of the modal dialog box. If `title` is an empty string (" "), the value of `prompt` is used as the title.
- `helpkey` — Specifies a string for a MULTI help key.
- `hostWin` — Specifies the host window from which the dialog box is shown. If the argument is `None` or if it is not a `GHS_Window` object, a window is randomly selected as the host window.

ChooseWindowFromGui()

```
ChooseWindowFromGui(msg="Choose a window:",  
title="Choose Window from List", wins=None,  
hostWin=None)
```

Allows you to choose a window from a window list displayed in a modal dialog box. This function returns an object for the chosen window, or it returns `None` upon failure or cancellation.

Arguments are:

- `msg` — Specifies the prompt string to display.

- `title` — Specifies the title of the modal dialog box. If `title` is an empty string (""), the value of `msg` is used as the title.
- `wins` — Specifies a list of windows from which you can choose. If you do not specify a window list or if `wins` is an empty string (""), the current MULTI IDE windows in the system are used.
- `hostWin` — Specifies the host window from which the dialog box is shown. If the argument is `None` or if it is not a `GHS_Window` object, a window is randomly selected as the host window.

Aliases are: `ChooseWindow()`, `ChooseWin()`

ChooseYesNo()

```
ChooseYesNo(msg, dftChoice=0, printOutput=True,  
hostWin=None)
```

Displays the specified message in a dialog box that prompts you to choose between **Yes** and **No**. This function returns `True` for **Yes** and `False` for **No**.

Arguments are:

- `msg` — Specifies the prompt message to display. The message should be a yes/no question.
- `dftChoice` — If 0, the default choice is **No**. If 1, the default choice is **Yes**.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.
- `hostWin` — Specifies the host window from which the dialog box is shown. If the argument is `None` or if it is not a `GHS_Window` object, a window is randomly selected as the host window.

Aliases are: `YesOrNo()`, `YesNo()`

GetInput()

```
GetInput(dftValue="", valList=[], onlyFromList=False,  
prompt="Your input", title="", helpkey="",  
hostWin=None)
```

Gets user input via a modal dialog box. This function returns the input string, or it returns an empty string ("") upon failure or cancellation.

Arguments are:

- `dftValue` — Specifies the default value to return.
- `valList` — Specifies a list of pre-defined values to include in the modal dialog box.
- `onlyFromList` — If `True`, you can only choose from the list of pre-defined values. If `False`, you can choose from the list of pre-defined values, or you can enter your own value.
- `prompt` — Specifies the prompt string to display.
- `title` — Specifies the title of the modal dialog box. If `title` is an empty string (""), the value of `prompt` is used as the title.
- `helpkey` — Specifies a string for a MULTI help key.
- `hostWin` — Specifies the host window from which the dialog box is shown. If the argument is `None` or if it is not a `GHS_Window` object, a window is randomly selected as the host window.

ShowMessage()

```
ShowMessage(msg, hostWin=None)
```

Displays the specified message in a dialog box and returns `True` on success and `False` on failure.

Arguments are:

- `msg` — Specifies the message to display.
- `hostWin` — Specifies the host window from which the dialog box is shown. If the argument is `None` or if it is not a `GHS_Window` object, a window is randomly selected as the host window.

Aliases are: `ShowMsg()`, `DisplayMessage()`, `DisplayMsg()`

GHS_WindowRegister Window Manipulation Functions

The following sections describe the window manipulation functions from class `GHS_WindowRegister`.

CloseAllWindows()

`CloseAllWindows()`

Closes all windows and returns `True` on success and `False` on failure.

Aliases are: `CloseWindows()`, `CloseWins()`

IconifyAllWindows()

`IconifyAllWindows()`

Minimizes all windows and returns `True` on success and `False` on failure.

Aliases are: `IconifyWindows()`, `IconifyWins()`, `IconWins()`, `MinimizeWindows()`, `MinWins()`

RestoreAllWindows()

`RestoreAllWindows()`

Restores all minimized windows and returns `True` on success and `False` on failure.

Aliases are: `RestoreWindows()`, `RestoreWins()`

GHS_WindowRegister Wait Functions

The following sections describe the functions from class `GHS_WindowRegister` that wait for some specified action.

WaitForWindow()

```
WaitForWindow(oldWinListInfo, duration=0.0,  
winClass="", winRegSvcName="", winName="", winId=0,  
warnIfNotFound=False)
```

Waits for the specified MULTI IDE window to appear and register. This function returns a `GHS_Window` object for the given window, or it returns `None` if the window does not register before the specified timeout.

Arguments are:

- `oldWinListInfo` — Specifies a window list that the window is not in.
- `duration` — Specifies how long the function waits, if at all. The `duration` may be:
 - A negative number — Indicates that the function waits for the expected window until the window appears.
 - `0.0` — Indicates that the function does not wait.
 - A positive number — Specifies the maximum number of seconds that the function waits.
- `winClass` — Specifies the class of the window. An empty string ("") matches any window class.
- `winRegSvcName` — Specifies the internal ID of the component to which the window belongs. An empty string ("") matches any internal ID.
- `winName` — Specifies the name of the window. An empty string ("") matches any window name.
- `winId` — Specifies the internal ID of the window. A 0 (zero) matches any window ID.
- `warnIfNotFound` — If `True` and if the specified window does not show up before the timeout, prints a warning message. If `False`, does not print a warning message.

Aliases are: `WaitWindow()`, `WaitWin()`

WaitForWindowFromClass()

```
WaitForWindowFromClass(oldWinListInfo, duration=0.0,  
winClass="", winRegSvcName="", warnIfNotFound=False)
```

Waits for a MULTI IDE window from a certain class to show up and register. This function returns a `GHS_Window` object for the given window, or it returns `None` if the window does not register before the specified timeout.

This function is a wrapper of `WaitForWindow()`. For argument descriptions, see “`WaitForWindow()`” on page 207.

The alias is: `WaitWinFromClass()`

WaitForWindowGoAway()

```
WaitForWindowGoAway(duration, winName,  
winId=0, winClass="", winRegSvcName="", pid=0,  
notInWinList=None)
```

Waits for the specified MULTI IDE window to remove registration and disappear. This function returns `True` if the specified MULTI IDE window does not exist (that is, registration has been removed) when the function returns, and `False` otherwise.

Arguments are:

- `duration` — Specifies how long the function waits, if at all. The `duration` may be:
 - A negative number — Indicates that the function waits until the window disappears.
 - `0.0` — Indicates that the function does not wait.
 - A positive number — Specifies the maximum number of seconds that the function waits.
- `winName` — Specifies the name of the window. This may be a regular expression. An empty string (`""`) matches any window name.

- `winId` — Specifies the internal ID of the window. A 0 (zero) matches any window ID.
- `winClass` — Specifies the class of the window. An empty string ("") matches any window class.
- `winRegSvcName` — Specifies the internal ID of the component to which the window belongs. An empty string ("") matches any internal ID.
- `pid` — Specifies the PID of the process to which the window belongs. A 0 (zero) matches any PID.
- `notInWinList` — Specifies a window list that does not contain the window you are waiting for. Windows in the list are ignored when this function attempts to determine whether the specified window has disappeared.

The alias is: `WaitForWinGoAway()`

WaitForWindowObjectGoAway()

```
WaitForWindowObjectGoAway(winObj, duration=0.0,  
notInWinList=None)
```

Waits for the specified MULTI IDE window to remove registration and disappear. This function returns `True` if the specified MULTI IDE window does not exist (that is, registration has been removed) when the function returns, and `False` otherwise.

This function is a wrapper of `WaitForWindowGoAway()` (see “`WaitForWindowGoAway()`” on page 208).

Arguments are:

- `winObj` — Specifies the window object for the window to be checked.
- `duration` — Specifies how long the function waits, if at all. The `duration` may be:
 - A negative number — Indicates that the function waits until the window disappears.
 - `0.0` — Indicates that the function does not wait.
 - A positive number — Specifies the maximum number of seconds that the function waits.

- `notInWinList` — Specifies a window list that does not contain the window you are waiting for. Windows in the list are ignored when this function attempts to determine whether the specified window has disappeared.

The alias is: `WaitForWinObjGoAway()`

Connection Functions

Chapter 11

This Chapter Contains:

- GHS_DebuggerApi Target Connection Functions
- GHS_DebuggerApi Window Display Functions
- GHS_DebugServer Functions
- GHS_Terminal Functions
- GHS_TerminalWindow Functions

This chapter documents functions from the following classes:

- `GHS_DebuggerApi` — Exposes the functions of the MULTI Debugger. This abstract class inherits from class `GHS_IdeObject`.

The `GHS_DebuggerApi` functions are divided into the following sections:

- “`GHS_DebuggerApi` Target Connection Functions” on page 213
- “`GHS_DebuggerApi` Window Display Functions” on page 216



Note This chapter documents only a subset of the functions from class `GHS_DebuggerApi`. For information about other `GHS_DebuggerApi` functions, see Chapter 12, “Debug Functions”.

- `GHS_DebugServer` — Exposes the functions of the MULTI debug server. This class inherits from class `GHS_Window` and class `GHS_DebuggerApi` because a Task Manager appears for run-mode debug connections. See “`GHS_DebugServer` Functions” on page 216.
- `GHS_Terminal` (alias `ghs_terminal`) — Exposes the basic functions of the MULTI Serial Terminal service. This class inherits from class `GHS_IdeObject`. See “`GHS_Terminal` Functions” on page 219.
- `GHS_TerminalWindow` — Exposes additional functions of the MULTI Serial Terminal. This class inherits from class `GHS_Window`. See “`GHS_TerminalWindow` Functions” on page 220.

GHS_DebuggerApi Target Connection Functions

The following sections describe the target connection functions from class `GHS_DebuggerApi`.

ConnectToRtserv()

```
ConnectToRtserv(target="", setupScript="",  
setupScriptArgs="", multiLog="", rtservLog="",  
moreOpts="", printOutput=True)
```

Connects to an RTOS target with the **rtserv** debug server. On success, this function returns a `GHS_DebugServer` object for the **rtserv** run-mode debug connection. Upon failure, it returns `None`.

Arguments are:

- `target` — Specifies the name of the target running the RTOS.
- `setupScript` — Specifies the board setup or connection initialization script filename.
- `setupScriptArgs` — Specifies arguments to the setup script. At present, only Python setup scripts can accept arguments.
- `multiLog` — Specifies a file with which to log the communication between MULTI and the debug server.
- `rtservLog` — Specifies a file with which to log the communication between the debug server and the RTOS target.
- `moreOpts` — Specifies other options for the debug server.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

Aliases are: `ConnectRtserv()`, `Rtserv()`

ConnectToRtserv2()

```
ConnectToRtserv2(target="", setupScript="",
setupScriptArgs="", multiLog="", rtservLog="",
moreOpts="", printOutput=True)
```

Connects to an RTOS target with the **rtserv2** debug server. On success, this function returns a `GHS_DebugServer` object for the **rtserv2** run-mode debug connection. Upon failure, it returns `None`.

For argument descriptions, see “ConnectToRtserv()” on page 213.

Aliases are: `ConnectRtserv2()`, `Rtserv2()`

ConnectToTarget()

```
ConnectToTarget(dbserver, setupScript="",
setupScriptArgs="", multiLog="",
stickToTheDebugger=True, moreOpts="",
printOutput=True)
```

Connects to a target with the specified debug server and options. On success, this function returns a `GHS_DebugServer` object for the established debug server connection. Upon failure, it returns `None`.

Arguments are:

- `dbserver` — Specifies the name of the debug server.
- `setupScript` — Specifies the board setup or connection initialization script filename.
- `setupScriptArgs` — Specifies arguments to the setup script. At present, only Python setup scripts can accept arguments.
- `multiLog` — Specifies a file with which to log the communication between MULTI and the debug server.
- `stickToTheDebugger` — If `True`, associates the established debug server connection with the current MULTI Debugger process. If `False`, does not create an association.
- `moreOpts` — Specifies other options for the debug server.

- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

Aliases are: `ConnectTarget()`, `Target()`, `Connect()`

Disconnect()

`Disconnect(printOutput=True)`

Disconnects the debug server connection associated with the current MULTI Debugger process. This function returns `True` on success and `False` on failure.

The argument is:

- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

IsConnected()

`IsConnected()`

Checks whether any debug server connection is established for the current MULTI Debugger process and returns `True` if yes and `False` otherwise.

The alias is: `Connected()`

GHS_DebuggerApi Window Display Functions

The following section describes one of the window display functions from class `GHS_DebuggerApi`. Additional `GHS_DebuggerApi` window display functions are documented in Chapter 12, “Debug Functions”.

ShowConnectionOrganizerWindow()

```
ShowConnectionOrganizerWindow(hardwareRegistryServer="",  
printOutput=True)
```

Displays the **Connection Organizer**. On success, this function returns a `GHS_ConnectionOrganizerWindow` object. Upon failure, it returns `None`.

Arguments are:

- `hardwareRegistryServer` — This argument has no effect. It is included for backwards compatibility.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

Aliases are: `ShowConnectionOrganizer()`, `ShowCo()`

GHS_DebugServer Functions

The following sections describe functions from class `GHS_DebugServer`.

__init__()

```
__init__(component, winName="", winId="",  
winClassName="", winRegSvcName="")
```

Initializes the object attributes.

Arguments are:

- `component` — Stores the identifier string for the Debugger component.

- `winName` — Stores the name that is registered for the window. This name may not be the same as the name shown on the window's title bar.
- `winId` — Stores the window's internal ID.
- `winClassName` — Stores the window's class. For a list of window classes, see "MULTI-Python Window Classes" on page 35.
- `winRegSvcName` — Stores the internal name of the object that registers the window. This attribute is for MULTI's internal use.

Disconnect()

```
Disconnect (printOutput=True)
```

Disconnects the debug server connection and returns `True` on success and `False` on failure.

The argument is:

- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

GetComponent()

```
GetComponent ()
```

Gets the MULTI Debugger component ID for the debug server connection. This function returns the debug server connection's component ID.

LoadProgram()

```
LoadProgram (progName="", block=True)
```

Loads a dynamic download module to the target if this feature is supported and if the target was configured with a dynamic loader (for example, the `LoaderTask` on `INTEGRITY`). This function returns `True` on success and `False` on failure.

Arguments are:

- `progName` — Specifies the program name.

- `block` — If `True`, executes `LoadProgram()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

Aliases are: `LoadProg()`, `LoadModule()`, `Load()`

RunCommands()

`RunCommands(cmds, block=True, printOutput=True)`

Executes MULTI Debugger commands in the context of a run-mode Task Manager or a debug server connection. This function returns `True` on success and `False` on failure.

Arguments are:

- `cmds` — Specifies the MULTI Debugger commands to execute.
- `block` — If `True`, executes `RunCommands()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any) from the MULTI debug server. If `False`, does not print the output.

Aliases are: `RunCommand()`, `RunCmd()`, `RunCmds()`

ShowTaskManagerWindow()

`ShowTaskManagerWindow()`

In a run-mode debugging environment, displays the Task Manager (if any) for the debug server. In a freeze-mode debugging environment, displays the **OSA Explorer** if the target is halted.

Upon success in a run-mode environment, this function returns a `GHS_TaskManagerWindow` object. Upon success in a freeze-mode environment, it returns a `GHS_OsaWindow` object. Upon failure, it returns `None`.

Aliases are: `ShowTaskWindow()`, `ShowTaskWin()`, `TaskWindow()`, `TaskWin()`

GHS_Terminal Functions

The following sections describe functions from class `GHS_Terminal`.

`__init__()`

```
__init__(workingDir=" ")
```

Initializes the object attributes.

The argument is:

- `workingDir` — Stores the working directory of the MULTI service object.

`MakeConnection()`

```
MakeConnection(connectCommand=" ")
```

Connects to a serial port. On success, this function returns a `GHS_TerminalWindow` object for the MTerminal window. Upon failure, it returns `None`.

The argument is:

- `connectCommand` — Specifies the command for connecting to a serial port. If `connectCommand` is an empty string (`" "`), a dialog box prompts you to select or create a serial terminal connection.

The alias is: `Connect()`

GHS_TerminalWindow Functions

The following sections describe functions from class `GHS_TerminalWindow`.

ChangeBaudRate()

`ChangeBaudRate(value, block=False)`

Changes the current baud rate and returns `True` on success and `False` on failure.

Arguments are:

- `value` — Specifies the value for the new baud rate.
- `block` — If `True`, executes `ChangeBaudRate()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

Connect()

`Connect(block=False)`

Connects to a serial port and returns `True` on success and `False` on failure.

The argument is:

- `block` — If `True`, executes `Connect()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

Disconnect()

`Disconnect(block=False)`

Disconnects the current connection and returns `True` on success and `False` on failure.

The argument is:

- `block` — If `True`, executes `Disconnect()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

SendBreak()

`SendBreak(block=False)`

Sends a break to the serial port. This function returns `True` on success and `False` on failure.

The argument is:

- `block` — If `True`, executes `SendBreak()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

Debug Functions

Chapter 12

This Chapter Contains:

- GHS_Debugger Functions
- GHS_DebuggerApi Debug Flag Functions
- GHS_DebuggerApi Host Information Functions
- GHS_DebuggerApi Memory Access Functions
- GHS_DebuggerApi Run-Control Attributes and Functions
- GHS_DebuggerApi Symbol Functions
- GHS_DebuggerApi Target Information Functions
- GHS_DebuggerApi Window Display Functions
- GHS_DebuggerWindow Basic Functions
- GHS_DebuggerWindow Breakpoint Functions
- GHS_DebuggerWindow Print Functions
- GHS_MemorySpaces Attributes and Functions
- GHS_OsTypes Attributes and Functions
- GHS_TargetIds Functions
- GHS_Task Basic Functions
- GHS_Task Run-Control Functions
- GHS_TraceWindow Functions

This chapter documents functions from the following classes:

- `GHS_Debugger` (alias `ghs_debugger`) — Exposes the basic functions of the MULTI Debugger service. This class inherits from class `GHS_DebuggerApi`. See “GHS_Debugger Functions” on page 225.
- `GHS_DebuggerApi` — Exposes the functions of the MULTI Debugger. This abstract class inherits from class `GHS_IdeObject`.

The `GHS_DebuggerApi` functions are divided into the following sections:

- “GHS_DebuggerApi Debug Flag Functions” on page 226
- “GHS_DebuggerApi Host Information Functions” on page 235
- “GHS_DebuggerApi Memory Access Functions” on page 236
- “GHS_DebuggerApi Run-Control Attributes and Functions” on page 238
- “GHS_DebuggerApi Symbol Functions” on page 244
- “GHS_DebuggerApi Target Information Functions” on page 245
- “GHS_DebuggerApi Window Display Functions” on page 249



Note Target connection functions and additional window display functions of class `GHS_DebuggerApi` are documented in Chapter 11, “Connection Functions”.

- `GHS_DebuggerWindow` — Exposes the functions of the MULTI Debugger. This class inherits from class `GHS_Window` and class `GHS_DebuggerApi`.

The `GHS_DebuggerWindow` functions are divided into the following sections:

- “GHS_DebuggerWindow Basic Functions” on page 250
- “GHS_DebuggerWindow Breakpoint Functions” on page 251
- “GHS_DebuggerWindow Print Functions” on page 254
- `GHS_MemorySpaces` — Defines the general memory space IDs used in the MULTI Debugger and debug servers. See “GHS_MemorySpaces Attributes and Functions” on page 255.
- `GHS_OsTypes` — Defines the operating system IDs supported in the MULTI Debugger. See “GHS_OsTypes Attributes and Functions” on page 256.

- `GHS_TargetIds` — Defines the target IDs supported in the MULTI Debugger. See “GHS_TargetIds Functions” on page 257.
- `GHS_Task` (alias `ghs_task`) — Exposes the functions of task debugging in an RTOS run-mode environment. Before a task is attached to, the object does not have `GHS_Window` attributes. However, you can still run some MULTI Debugger commands (such as run-control commands) on the object if the corresponding debug server supports them for unattached tasks.

This class inherits from class `GHS_DebuggerWindow`.

The `GHS_Task` functions are divided into the following sections:

- “GHS_Task Basic Functions” on page 258
- “GHS_Task Run-Control Functions” on page 260
- `GHS_TraceWindow` — Exposes the special functions of the MULTI Debugger’s Trace List. This class inherits from class `GHS_Window`. See “GHS_TraceWindow Functions” on page 263.

GHS_Debugger Functions

The following sections describe functions from class `GHS_Debugger`.

`__init__()`

`__init__(workingDir="")`

Initializes the object attributes.

The argument is:

- `workingDir` — Stores the working directory of the MULTI service object.

RunCommands()

```
RunCommands(cmds, block=True, printOutput=True)
```

Executes the specified MULTI Debugger commands and returns `True` on success and `False` on failure.

Arguments are:

- `cmds` — Specifies the MULTI Debugger commands to execute.
- `block` — If `True`, executes `RunCommands()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

Aliases are: `RunCommand()`, `RunCmd()`, `RunCmds()`, `ExecuteCmd()`, `ExecuteCmds()`, `ExecCmds()`, `ExecCmd()`

GHS_DebuggerApi Debug Flag Functions

The following sections describe the functions from class `GHS_DebuggerApi` that relate to debug flags.

ChangeBreakpointInheritance()

```
ChangeBreakpointInheritance(toggle=True,  
newStatus=False)
```

Enables or disables the option:

Inherit Software Breakpoint After Forking

for the current MULTI Debugger process. This option corresponds to the **P b** command. For more information, see the **P** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

This function returns `True` on success and `False` on failure.

Arguments are:

- `toggle` — If `True`, toggles the setting for the option, and ignores the new status. If `False`, does not toggle the setting.
- `newStatus` — If `True`, specifies that the option's new status is `On`. If `False`, specifies that the option's new status is `Off`.

Aliases are: `ChangeBpInheritance()`, `ChangeB()`

ChangeDebugChildren()

`ChangeDebugChildren(toggle=True, newStatus=False)`

Enables or disables the option:

`Debug Child Tasks/Processes`

for the current `MULTI` Debugger process. This option corresponds to the **P c** command. For more information, see the **P** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

This function returns `True` on success and `False` on failure.

Arguments are:

- `toggle` — If `True`, toggles the setting for the option, and ignores the new status. If `False`, does not toggle the setting.
- `newStatus` — If `True`, specifies that the option's new status is `On`. If `False`, specifies that the option's new status is `Off`.

The alias is: `ChangeC()`

ChangeDebugOnTaskCreation()

`ChangeDebugOnTaskCreation(toggle=True, newStatus=False)`

Enables or disables the option:

`Debug on Task Creation`

for the current MULTI Debugger process. This option corresponds to the **P d** command. For more information, see the **P** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

This function returns `True` on success and `False` on failure.

Arguments are:

- `toggle` — If `True`, toggles the setting for the option, and ignores the new status. If `False`, does not toggle the setting.
- `newStatus` — If `True`, specifies that the option’s new status is `On`. If `False`, specifies that the option’s new status is `Off`.

The alias is: `Changed()`

ChangeHaltOnAttach()

```
ChangeHaltOnAttach(toggle=True, newStatus=False)
```

Enables or disables the option:

Halt on Attach

for the current MULTI Debugger process. This option corresponds to the **P h** command. For more information, see the **P** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

This function returns `True` on success and `False` on failure.

Arguments are:

- `toggle` — If `True`, toggles the setting for the option, and ignores the new status. If `False`, does not toggle the setting.
- `newStatus` — If `True`, specifies that the option’s new status is `On`. If `False`, specifies that the option’s new status is `Off`.

The alias is: `ChangeH()`

ChangeInheritProcessBits()

```
ChangeInheritProcessBits (toggle=True,  
newStatus=False)
```

Enables or disables the option:

Inherit Process Bits in Child

for the current MULTI Debugger process. This option corresponds to the **P i** command. For more information, see the **P** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

This function returns `True` on success and `False` on failure.

Arguments are:

- `toggle` — If `True`, toggles the setting for the option, and ignores the new status. If `False`, does not toggle the setting.
- `newStatus` — If `True`, specifies that the option’s new status is `On`. If `False`, specifies that the option’s new status is `Off`.

The alias is: `ChangeI()`

ChangeRunOnDetach()

```
ChangeRunOnDetach (toggle=True, newStatus=False)
```

Enables or disables the option:

Run on Detach

for the current MULTI Debugger process. This option corresponds to the **P r** command. For more information, see the **P** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

This function returns `True` on success and `False` on failure.

Arguments are:

- `toggle` — If `True`, toggles the setting for the option, and ignores the new status. If `False`, does not toggle the setting.
- `newStatus` — If `True`, specifies that the option's new status is `On`. If `False`, specifies that the option's new status is `Off`.

The alias is: `ChangeR()`

ChangeStopAfterExec()

```
ChangeStopAfterExec(toggle=True, newStatus=False)
```

Enables or disables the option:

`Stop After Exec`

for the current MULTI Debugger process. This option corresponds to the **P e** command. For more information, see the **P** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

This function returns `True` on success and `False` on failure.

Arguments are:

- `toggle` — If `True`, toggles the setting for the option, and ignores the new status. If `False`, does not toggle the setting.
- `newStatus` — If `True`, specifies that the option's new status is `On`. If `False`, specifies that the option's new status is `Off`.

The alias is: `ChangeE`

ChangeStopAfterFork()

```
ChangeStopAfterFork(toggle=True, newStatus=False)
```

Enables or disables the option:

`Stop After Fork`

for the current MULTI Debugger process. This option corresponds to the **P f** command. For more information, see the **P** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

This function returns `True` on success and `False` on failure.

Arguments are:

- `toggle` — If `True`, toggles the setting for the option, and ignores the new status. If `False`, does not toggle the setting.
- `newStatus` — If `True`, specifies that the option’s new status is `On`. If `False`, specifies that the option’s new status is `Off`.

The alias is: `ChangeF()`

ChangeStopOnTaskCreation()

```
ChangeStopOnTaskCreation(toggle=True,  
newStatus=False)
```

Enables or disables the option:

Stop on Task Creation

for the current MULTI Debugger process. This option corresponds to the **P t** command. For more information, see the **P** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

This function returns `True` on success and `False` on failure.

Arguments are:

- `toggle` — If `True`, toggles the setting for the option, and ignores the new status. If `False`, does not toggle the setting.
- `newStatus` — If `True`, specifies that the option’s new status is `On`. If `False`, specifies that the option’s new status is `Off`.

The alias is: `ChangeT()`

CheckBreakpointInheritance()

`CheckBreakpointInheritance()`

Checks if the option `Inherit Software Breakpoint After Forking` is enabled for the current MULTI Debugger process. This option corresponds to the **P b** command. For more information, see the **P** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

This function returns `True` if the option is enabled and `False` otherwise.

Aliases are: `CheckBpInheritance()`, `CheckB()`

CheckDebugChildren()

`CheckDebugChildren()`

Checks if the option `Debug Child Tasks/Processes` is enabled for the current MULTI Debugger process. This option corresponds to the **P c** command. For more information, see the **P** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

This function returns `True` if the option is enabled and `False` otherwise.

The alias is: `CheckC()`

CheckDebugOnTaskCreation()

`CheckDebugOnTaskCreation()`

Checks if the option `Debug on Task Creation` is enabled for the current MULTI Debugger process. This option corresponds to the **P d** command. For more information, see the **P** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

This function returns `True` if the option is enabled and `False` otherwise.

The alias is: `CheckD()`

CheckHaltOnAttach()

CheckHaltOnAttach()

Checks if the option `Halt on Attach` is enabled for the current MULTI Debugger process. This option corresponds to the **P h** command. For more information, see the **P** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

This function returns `True` if the option is enabled and `False` otherwise.

The alias is: `CheckH()`

CheckInheritProcessBits()

CheckInheritProcessBits()

Checks if the option `Inherit Process Bits in Child` is enabled for the current MULTI Debugger process. This option corresponds to the **P i** command. For more information, see the **P** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

This function returns `True` if the option is enabled and `False` otherwise.

The alias is: `CheckI()`

CheckRunOnDetach()

CheckRunOnDetach()

Checks if the option `Run on Detach` is enabled for the current MULTI Debugger process. This option corresponds to the **P r** command. For more information, see the **P** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

This function returns `True` if the option is enabled and `False` otherwise.

The alias is: `CheckR()`

CheckStopAfterExec()

`CheckStopAfterExec()`

Checks if the option `Stop After Exec` is enabled for the current MULTI Debugger process. This option corresponds to the **P e** command. For more information, see the **P** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

This function returns `True` if the option is enabled and `False` otherwise.

The alias is: `CheckE()`

CheckStopAfterFork()

`CheckStopAfterFork()`

Checks if the option `Stop After Fork` is enabled for the current MULTI Debugger process. This option corresponds to the **P f** command. For more information, see the **P** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

This function returns `True` if the option is enabled and `False` otherwise.

The alias is: `CheckF()`

CheckStopOnTaskCreation()

`CheckStopOnTaskCreation()`

Checks if the option `Stop on Task Creation` is enabled for the current MULTI Debugger process. This option corresponds to the **P t** command. For more information, see the **P** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.

This function returns `True` if the option is enabled and `False` otherwise.

The alias is: `CheckT()`

GHS_DebuggerApi Host Information Functions

The following sections describe the functions from class `GHS_DebuggerApi` that relate to host information.

GetHostOsName()

`GetHostOsName()`

Gets the host operating system name. This function returns a string for the host operating system name, or it returns an empty string (" ") upon error.

The alias is: `HostOsName()`

GetMultiVersion()

`GetMultiVersion(which=0)`

Gets MULTI's major, minor, or micro version and returns it. Versions are denoted in the following manner: major.minor.micro. For example, MULTI 5.0.1 indicates major version 5 of MULTI, minor version 0 of MULTI 5, and micro version 1 of minor version 0.

The argument is:

- `which` — Specifies the version to return. You may specify:
 - 0 — Indicates MULTI's major version.
 - 1 — Indicates MULTI's minor version.
 - 2 — Indicates MULTI's micro version.

The alias is: `MultiVersion()`

GHS_DebuggerApi Memory Access Functions

The following sections describe the memory access functions from class GHS_DebuggerApi.

ReadIndirectValue()

```
ReadIndirectValue(addressPort, addressValue,  
valuePort, addressSizeInBytes=4, valueSizeInBytes=4,  
printError=True)
```

Indirectly reads an integer from memory by writing a value to a memory location (addressPort) and then reading the value from another memory location (valuePort). This function returns the integer from memory, or it returns 0 upon error.

Arguments are:

- addressPort — Specifies the memory location where the value is written.
- addressValue — Specifies the value that is written to addressPort.
- valuePort — Specifies the memory location from which to read the value.
- addressSizeInBytes — Specifies the size of the value written to addressPort.
- valueSizeInBytes — Specifies the size of the value read from valuePort.
- printError — If True, prints error messages (if any). If False, does not print error messages.

Aliases are: ReadIndirectInteger(), ReadIndirectInt(), ReadIndInt()

ReadIntegerFromMemory()

```
ReadIntegerFromMemory(address, sizeInBytes=4,  
printError=True)
```

Reads an integer from the specified memory location and returns the integer from memory or returns 0 upon failure.

Arguments are:

- `address` — Specifies the memory location.
- `sizeInBytes` — Specifies the size (in bytes) of the integer value. The supported sizes are: 1, 2 and 4.
- `printError` — If `True`, prints error messages (if any). If `False`, does not print error messages.

The alias is: `ReadInt()`

ReadStringFromMemory()

```
ReadStringFromMemory(address, printError=True)
```

Reads a null-terminated string from the specified memory location and returns the string from memory or returns an empty string (" ") upon failure.

Arguments are:

- `address` — Specifies the memory location.
- `printError` — If `True`, prints error messages (if any). If `False`, does not print error messages.

The alias is: `ReadStr`

WriteIntegerToMemory()

```
WriteIntegerToMemory(address, value, sizeInBytes=4,  
printError=True)
```

Writes an integer to the specified memory location and returns `True` on success and `False` on failure.

Arguments are:

- `address` — Specifies the memory location.
- `value` — Specifies the integer value to write.
- `sizeInBytes` — Specifies the size (in bytes) of `value`. The supported sizes are: 1, 2 and 4.

- `printError` — If `True`, prints error messages (if any). If `False`, does not print error messages.

The alias is: `WriteInt()`

WriteStringToMemory()

```
WriteStringToMemory(address, stringValue,  
printError=True)
```

Writes a string to the specified memory location and returns `True` on success and `False` on failure.

Arguments are:

- `address` — Specifies the memory location.
- `stringValue` — Specifies the string value to write.
- `printError` — If `True`, prints error messages (if any). If `False`, does not print error messages.

Aliases are: `WriteString()`, `WriteStr()`

GHS_DebuggerApi Run-Control Attributes and Functions

The following list describes the attributes of this class:

- `status_nil` — Indicates that no program is loaded in the MULTI Debugger.
- `status_no_process` — Indicates that the program is not loaded on the target.
- `status_stopped` — Indicates that the program is stopped on the target.
- `status_running` — Indicates that the program is running on the target.
- `status_dying` — Indicates that the program is dying.
- `status_forking` — Indicates that the program is forking another process.
- `status_executing` — Indicates that the program just finished a single-step operation.

- `status_continuing` — Indicates that the program is about to continue execution.
- `status_zombie` — Indicates that the program is zombied (that is, it has exited by calling `exit()`, but data structures describing the program still exist on the target).

The following sections describe the run-control functions from class `GHS_DebuggerApi`.

DebugProgram()

```
DebugProgram(fileName, newWin=True, block=True,  
printOutput=True, expandFileName=True)
```

Loads a program in the Debugger, replacing the currently selected target list entry or adding a new target list entry. On success, this function returns a `GHS_DebuggerWindow` object for the MULTI Debugger window. On failure, it returns `None`.

Arguments are:

- `fileName` — Specifies the program to be loaded in the Debugger.
- `newWin` — If `True`, adds a new target list entry. If `False`, replaces the currently selected target list entry.
- `block` — If `True`, executes `DebugProgram()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.
- `expandFileName` — If `True`, uses the client's current working directory to expand `fileName` into a complete file path. If `False`, directly transfers `fileName` to the corresponding service, which resolves the filename with its current working directory if necessary. For more information, see the `expandFileName` description in “`LoadWorkspaceFile()`” on page 305.

Aliases are: `DebugProg()`, `Debug()`, `DebugFile()`

GetPc()

`GetPc()`

Gets the PC value of the program being debugged. On success, this function returns the PC register's value. On error, it returns -1.

The alias is: `Pc()`

GetProgram()

`GetProgram()`

Gets the name of the program being debugged and returns a string for the name, or returns an empty string (" ") upon failure.

Aliases are: `DebuggedProgram()`, `ProgName()`

GetStatus()

`GetStatus()`

Gets the MULTI process's status as a number and returns the number or returns -1 upon error.

GetTargetPid()

`GetTargetPid()`

Gets the ID of the process being debugged on the target. This function returns the ID, or it returns 0 upon error.

The alias is: `TargetPid()`

Halt()

`Halt(block=True, printOutput=True)`

Halts the process currently being debugged on the target and returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `Halt()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

The alias is: `Stop()`

IsHalted()

`IsHalted()`

Checks whether the process currently being debugged is halted on the target and returns `True` if yes, and `False` otherwise.

The alias is: `IsStopped()`

IsRunning()

`IsRunning()`

Checks whether the process currently being debugged is running on the target and returns `True` if yes, and `False` otherwise.

IsStarted()

`IsStarted()`

Checks whether the program being debugged has been loaded and started on the target. This function returns `True` if yes, and `False` otherwise.

The alias is: `HasChild()`

Kill()

```
Kill(force=True, block=True, printOutput=True)
```

Kills the current process being debugged on the target. This function returns `True` on success and `False` on failure.

Arguments are:

- `force` — If `True`, forcefully kills the process. If `False`, does not forcefully kill the process.
- `block` — If `True`, executes `Kill()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

Next()

```
Next(block=True, printOutput=True,  
stepIntoFunc=False)
```

Single-steps the process currently being debugged on the target, stepping over function calls by default. This function returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `Next()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.
- `stepIntoFunc` — If `True`, steps into functions when the current position is a function call. If `False`, does not step into functions.

Resume()

`Resume(block=True, printOutput=True)`

Resumes the process currently being debugged on the target. This function returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `Resume()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

The alias is: `Run()`

Step()

`Step(block=True, printOutput=True, stepIntoFunc=True)`

Single-steps the process currently being debugged on the target, stepping into function calls by default. This function returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `Step()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.
- `stepIntoFunc` — If `True`, steps into functions when the current position is a function call. If `False`, does not step into functions.

WaitToStop()

```
WaitToStop(duration=-1.0, checkInterval=0.5)
```

Waits for the process being debugged to stop on the target. This function returns `True` if the process has stopped before the timeout, and it returns `False` otherwise.

Arguments are:

- `duration` — Specifies how long the function waits, if at all. The `duration` may be:
 - A negative number — Indicates that the function waits until the process is stopped on the target.
 - 0 — Indicates that the function does not wait at all.
 - A positive number — Specifies the maximum number of seconds that the function waits.
- `checkInterval` — Specifies how often the status of the process is checked.

GHS_DebuggerApi Symbol Functions

The following sections describe the functions from class `GHS_DebuggerApi` that relate to symbols.

CheckSymbol()

```
CheckSymbol(symbolName=" ")
```

Checks whether a symbol exists in the MULTI Debugger process and returns `True` if yes and `False` otherwise.

The argument is:

- `symbolName` — Specifies the name of the symbol.

The alias is: `CheckSym()`

GetSymbolAddress()

`GetSymbolAddress (symbolName=" ")`

Gets the memory address of a symbol in the MULTI Debugger process. This function returns the symbol's memory address, or it returns 0 upon error.

The argument is:

- `symbolName` — Specifies the name of the symbol.

The alias is: `GetSymAdr()`

GetSymbolSize()

`GetSymbolSize (symbolName=" ")`

Gets a symbol's size if it exists. This function returns the size (in bytes) of the symbol, or it returns a 0 upon error.

The argument is:

- `symbolName` — Specifies the name of the symbol.

The alias is: `GetSymSize()`

GHS_DebuggerApi Target Information Functions

The following sections describe the functions from class `GHS_DebuggerApi` that relate to target information.

BigEndianTarget()

`BigEndianTarget()`

Checks whether the target is big endian or little endian. This function returns `True` if the target is big endian and `False` if it is little endian.

GetCpuFamily()

`GetCpuFamily(targetId=0)`

Gets the target CPU family ID from the target ID. This function returns the target CPU family ID (see “GHS_TargetIds Functions” on page 257), or it returns 0 upon failure.

The argument is:

- `targetId` — Specifies the target ID. If `targetId` is 0, the function gets the target ID by itself.

The alias is: `CpuFamily()`

GetCpuMinor()

`GetCpuMinor(targetId=0)`

Gets the target CPU minor ID. This function returns the target CPU minor ID (see “GHS_TargetIds Functions” on page 257), or it returns 0 upon failure.

The argument is:

- `targetId` — Specifies the target ID. If `targetId` is 0, the function gets the target ID by itself.

The alias is: `CpuMinor()`

GetTargetCoProcessor()

`GetTargetCoProcessor()`

Gets the target coprocessor ID and returns the ID (see “GHS_TargetIds Functions” on page 257) or returns 0 upon failure.

The alias is: `CoProcessor()`

GetTargetCpuFamilyName()

GetTargetCpuFamilyName()

Gets the target CPU family name, which is the same as the name used in the MULTI project file. This function returns the target CPU family name, or it returns an empty string (" ") upon failure.

Aliases are: GetCpuFamilyName(), GetCpuName(), CpuName()

GetTargetId()

GetTargetId()

Gets the target ID. This function returns the ID, or it returns a 0 upon failure. See the description of _TARGET in “System Variables” in Chapter 12, “Using Expressions, Variables, and Procedure Calls” in the *MULTI: Debugging* book.

GetTargetOsMinorType()

GetTargetOsMinorType()

Gets the target operating system minor type. This function returns the minor type, or it returns a 0 upon error. For various minor types, see “GHS_OsTypes Attributes and Functions” on page 256.

Aliases are: GetTargetOsMinor(), OsMinor()

GetTargetOsName()

GetTargetOsName(detail=True)

Gets the target operating system name, if any. This function returns the target operating system name, or it returns "standalone" for stand-alone programs. The returned string is the same as the name used in the MULTI project file.

The argument is:

- detail — If True, gets the detailed OS name, if any (for example, linux). If False, gets the generic OS name (for example, unix).

Aliases are: `GetOsName()`, `OsName()`

GetTargetOsType()

`GetTargetOsType()`

Gets the target operating system type. This function returns the operating system type, or it returns 0 for stand-alone programs. For various operating system types, see “GHS_OsTypes Attributes and Functions” on page 256.

Aliases are: `GetTargetOs()`, `OsType()`

GetTargetSeries()

`GetTargetSeries()`

Gets the target series ID. This function returns the ID, or it returns 0 upon failure.

Aliases are: `GetSeries()`, `Series()`

IsFreezeMode()

`IsFreezeMode()`

Checks whether the MULTI Debugger is in a freeze-mode debugging environment and returns `True` if yes and `False` otherwise.

Aliases are: `IsStopMode()`, `StopMode()`

IsNativeDebugging()

`IsNativeDebugging()`

Checks whether the program being debugged is for native debugging and returns `True` if yes and `False` otherwise.

Aliases are: `NativeDebugging()`, `NativeProg()`

IsRunMode()

`IsRunMode()`

Checks whether the MULTI Debugger is in a run-mode debugging environment and returns `True` if yes and `False` otherwise.

Aliases are: `InRunMode()`, `RunMode()`

GHS_DebuggerApi Window Display Functions

The following sections describe window display functions from class `GHS_DebuggerApi`. Additional `GHS_DebuggerApi` window display functions are documented in Chapter 11, “Connection Functions”.

ShowOsaExplorerWindow()

`ShowOsaExplorerWindow(haltTargetIfNecessary=True, printOutput=True)`

Displays the **OSA Explorer**. On success, this function returns a `GHS_OsaWindow` object. Upon failure, it returns `None`.

Arguments are:

- `haltTargetIfNecessary` — If `True` and you are debugging in freeze mode, halts the target (launching the **OSA Explorer** in freeze mode requires that the target be halted via this argument setting). If `False` and you are debugging in freeze mode, this function fails. In run mode, this argument has no effect (the target is automatically halted if necessary).
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

Aliases are: `ShowOsaExplorer()`, `ShowOsa()`

ShowTaskManagerWindow()

ShowTaskManagerWindow()

In a run-mode debugging environment, displays the Task Manager (if any) for the debug server. In a freeze-mode debugging environment, displays the **OSA Explorer** if the target is halted.

Upon success in a run-mode environment, this function returns a GHS_TaskManagerWindow object. Upon success in a freeze-mode environment, it returns a GHS_OsaWindow object. Upon failure, it returns None.

Aliases are: ShowTaskWindow(), ShowTaskWin(), TaskWindow(), TaskWin()

ShowTraceWindow()

ShowTraceWindow(printOutput=True)

Displays the **Trace List**. On success, this function returns a GHS_TraceWindow object. Upon failure, it returns None.

The argument is:

- printOutput — If True, prints the output (if any). If False, does not print output.

The alias is: ShowTraceWin()

GHS_DebuggerWindow Basic Functions

The following section describes the function from class GHS_DebuggerWindow.

RunCommands()

RunCommands(cmds, block=True, printOutput=True)

Runs commands and returns True on success and False on failure.

Arguments are:

- `cmds` — Specifies the commands to be executed.
- `block` — If `True`, executes `RunCommands()` in blocked mode and grabs the output, if any. If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

Aliases are: `RunCommand()`, `RunCmd()`, `RunCmds()`

GHS_DebuggerWindow Breakpoint Functions

The following sections describe the breakpoint functions from class `GHS_DebuggerWindow`.

RemoveBreakpoint()

`RemoveBreakpoint(location="", grabOutput=False)`

Removes software breakpoint set at the specified location. If you do not specify a location, all software breakpoints set in the MULTI Debugger window are removed. This function returns `True` on success and `False` on failure.

Arguments are:

- `location` — Specifies the location of the breakpoint that is removed.
- `grabOutput` — If `True`, executes `RemoveBreakpoint()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.

Aliases are: `RemoveBp()`, `RmBp()`

SetBreakpoint()

```
SetBreakpoint(location="", bpType="",  
grabOutput=False)
```

Sets a software breakpoint at the specified location and returns `True` on success and `False` on failure.

Arguments are:

- `location` — Specifies the location where the breakpoint is set.
- `bpType` — Specifies a string such as `"at"` for the breakpoint type.
- `grabOutput` — If `True`, executes `SetBreakpoint()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.

The alias is: `SetBp()`

SetGroupBreakpoint()

```
SetGroupBreakpoint(location="", hitGrp="",  
haltGrp="", grabOutput=False)
```

Sets a group breakpoint at the specified location. This function returns `True` on success and `False` on failure.

Arguments are:

- `location` — Specifies the location where the breakpoint is set.
- `hitGrp` — Specifies the name of the group whose tasks are able to hit the breakpoint. If you do not specify a group, a group consisting only of the current task is used.
- `haltGrp` — Specifies the name of the group to halt. If you do not specify a group, a group consisting only of the current task is used.
- `grabOutput` — If `True`, executes `SetGroupBreakpoint()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.

The alias is: `SetGrpBp()`

ShowBreakpoints()

`ShowBreakpoints(block=True, printOutput=True)`

Shows information about all software breakpoints visible in the MULTI Debugger window. This function returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `ShowBreakpoints()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

The alias is: `ShowBps()`

ShowBreakpointWindow()

`ShowBreakpointWindow(block=True)`

Displays the **Breakpoints** window. If this function is executed successfully and the `block` argument is `True`, the function returns a `GHS_Window` object for the **Breakpoints** window. If this function fails or if the `block` argument is `False`, the function returns `None`.

The argument is:

- `block` — If `True`, executes `ShowBreakpointWindow()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.

The alias is: `ShowBpWin()`

GHS_DebuggerWindow Print Functions

The following sections describe the print functions from class `GHS_DebuggerWindow`.

DumpToFile()

```
DumpToFile(fileName="", block=True, printOutput=True)
```

Writes the contents of the Debugger's source pane into a text file. This function returns `True` on success and `False` on failure.

Arguments are:

- `fileName` — Specifies the name of the file to write to. If you do not specify a filename, the Debugger opens a file chooser from which you can select a file.
- `block` — If `True`, executes `DumpToFile()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

PrintFile()

```
PrintFile(block=True, printOutput=True)
```

Prints the entire source file currently being viewed in the Debugger window. This function returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `PrintFile()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

The alias is: `Print()`

PrintWindow()

```
PrintWindow(block=True, printOutput=True)
```

Prints the contents currently being viewed in the Debugger window. This function returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `PrintWindow()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

The alias is: `PrintWin()`

GHS_MemorySpaces Attributes and Functions

Class `GHS_MemorySpaces` defines the general memory space IDs used in the MULTI Debugger and in debug servers.

The following list describes the attributes of this class:

- `MSPACE_DEFAULT` — Stores the ID for the default memory space.
- `MSPACE_TEXT_DEFAULT` — Stores the ID for the code's default memory space.
- `MSPACE_DATA_DEFAULT` — Stores the ID for the data's default memory space.
- `MSPACE_TEXT_PHYSICAL` — Stores the ID for the code's physical memory space.
- `MSPACE_DATA_PHYSICAL` — Stores the ID for the data's physical memory space.
- `MSPACE_DATA_RAWMEMORY` — Stores the ID for a special memory space that instructs the MULTI Debugger to bypass all caches and read/write directly from/to memory.

The following section describes the function from class `GHS_MemorySpaces`.

__init__()

`__init__()`

Initializes object attributes.

GHS_OsTypes Attributes and Functions

Class `GHS_OsTypes` defines the operating system IDs supported in the MULTI Debugger.

The following list describes the attributes of this class:

- `OS_9K` — Stores the ID for the 9k real-time operating system.
- `OS_ALPHA` — Stores the ID for the Alpha operating system.
- `OS_GENERIC` — Stores the ID for generic real-time operating systems.
- `OS_INTEGRITY` — Stores the ID for the INTEGRITY real-time operating system.
- `OS_MERCURY` — Stores the ID for the Mercury real-time operating system.
- `OS_MINOR_NUL` — Stores the ID for the trivial minor OS type.
- `OS_MINOR_UNIX_LINUX` — Stores the ID for the UNIX operating system's minor type Linux.
- `OS_MINOR_UNIX_LYNXOS` — Stores the ID for the UNIX operating system's minor type LynxOS.
- `OS_MULTICORE` — Stores the ID for the general multi-core system.
- `OS_NUCLEUS` — Stores the ID for the Nucleus real-time operating system.
- `OS_NULL` — Stores the ID for stand-alone programs.
- `OS_THREADX` — Stores the ID for the ThreadX real-time operating system.
- `OS_UNIX` — Stores the ID for the UNIX operating system.
- `OS_VXWORKS` — Stores the ID for the VxWorks real-time operating system.
- `OS_WINDOWS` — Stores the ID for the Windows operating system.

The following section describes the function from class `GHS_OsTypes`.

`__init__()`

```
__init__()
```

Initializes object attributes.

GHS_TargetIds Functions

Class `GHS_TargetIds` defines the target IDs supported in the MULTI Debugger. About 500 such IDs exist. To view a complete list of target IDs and function names, enter the following statement:

```
Python> ghs_printobject(dir(targetIds))
```

You can use an ID to get its value. For example:

```
Python> hex(targetIds.XSCALE_IXP2350)
```

gets the value (in hex) for `XSCALE_IXP2350`.



Note The values are constant; do not change them.

The following section describes the function from class `GHS_TargetIds`.

`__init__()`

```
__init__()
```

Initializes object attributes.

GHS_Task Basic Functions

The following sections describe the basic functions from class GHS_Task.

`__init__()`

```
__init__(dbcomponent, addressSpace, taskIdOrName,
taskComponent="", wn="", wid="0", cn="",
regSvcName="")
```

Initializes the object attributes.

Arguments are:

- `dbcomponent` — Stores the component name of the debug server.
- `addressSpace` — Stores the name of the address space that contains the task.
- `taskIdOrName` — Stores the ID or the name of the task.
- `taskComponent` — Stores the component name of the task.
- `wn` — Stores the name of the Debugger window in which the task is being debugged.
- `wid` — Stores the ID of the Debugger window in which the task is being debugged.
- `cn` — Stores the class name of the Debugger window in which the task is being debugged.
- `regSvcName` — Stores the register service name of the Debugger window in which the task is being debugged.

`RunCommands()`

```
RunCommands(cmds, block=True, printOutput=True)
```

Executes MULTI Debugger commands via the corresponding Debugger window if the task is attached, or via the corresponding debug server otherwise. This function returns `True` on success and `False` on failure.

Arguments are:

- `cmds` — Specifies the MULTI Debugger commands.
- `block` — If `True`, executes `RunCommands()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

Aliases are: `RunCommand()`, `RunCmd()`, `RunCmds()`

RunCommandsViaDebugServer()

```
RunCommandsViaDebugServer(cmds, block=True,  
printOutput=True)
```

Executes MULTI Debugger commands via the corresponding debug server. This function returns `True` on success and `False` on failure.

Arguments are:

- `cmds` — Specifies the MULTI Debugger commands.
- `block` — If `True`, executes `RunCommandsViaDebugServer()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

Aliases are: `RunCmdViaDebugServer()`, `RunCmdViaDbserve()`

GHS_Task Run-Control Functions

The following sections describe the run-control functions from class `GHS_Task`.

Attach()

```
Attach(flags="", block=True, printOutput=True)
```

Attaches to the task and returns a string for the task's component ID.

Arguments are:

- `flags` — Specifies options to the MULTI Debugger **attach** command. For available options, see the **attach** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.
- `block` — If `True`, executes `Attach()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

Detach()

```
Detach(flags="", block=False, printOutput=True)
```

Detaches from the task and returns `True` on success and `False` on failure.

Arguments are:

- `flags` — Specifies options to the MULTI Debugger **detach** command. For available options, see the **detach** command in Chapter 3, “General Debugger Command Reference” in the *MULTI: Debugging Command Reference* book.
- `block` — If `True`, executes `Detach()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

Halt()

```
Halt(block=True, printOutput=True)
```

Halts execution of the task. If the task is not attached and the debug server does not support halting without attaching, the operation fails. This function returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `Halt()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

The alias is: `Stop()`

Next()

```
Next(block=True, printOutput=True,  
stepIntoFunc=False)
```

Single-steps the task, stepping over function calls by default. If the task is not attached and the debug server does not support resuming without attaching, the operation fails. This function returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `Next()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.
- `stepIntoFunc` — If `True`, steps into functions when the current position is a function call. If `False`, does not step into functions.

Resume()

```
Resume(block=True, printOutput=True)
```

Resumes execution of the task. If the task is not attached and the debug server does not support resuming without attaching, the operation fails. This function returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `Resume()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output. If `False`, does not print the output.

The alias is: `Run()`

Step()

```
Step(block=True, printOutput=True, stepIntoFunc=True)
```

Single-steps the task, stepping into function calls by default. If the task is not attached and the debug server does not support resuming without attaching, the operation fails. This function returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `Step()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.
- `stepIntoFunc` — If `True`, steps into functions when the current position is a function call. If `False`, does not step into functions.

GHS_TraceWindow Functions

The following sections describe functions from class `GHS_TraceWindow`.

FlushTraceBuffer()

`FlushTraceBuffer(block=True)`

Flushes the trace buffer on the target. This function returns `True` on success and `False` on failure.

The argument is:

- `block` — If `True`, executes `FlushTraceBuffer()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

The alias is: `FlushTraceBuf()`

JumpToTrigger()

`JumpToTrigger(block=True)`

Jumps to the trigger and returns `True` on success and `False` on failure.

The argument is:

- `block` — If `True`, executes `JumpToTrigger()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

StartTracing()

`StartTracing(block=True)`

Starts the collection of trace data and returns `True` on success and `False` on failure.

The argument is:

- `block` — If `True`, executes `StartTracing()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

Aliases are: `StartTrace()`, `TraceOn()`

StopTracing()

`StopTracing(block=True)`

Stops the collection of trace data and returns `True` on success and `False` on failure.

The argument is:

- `block` — If `True`, executes `StopTracing()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

Aliases are: `StopTrace()`, `TraceOff()`

Editor Functions

Chapter 13

This Chapter Contains:

- GHS_Editor Functions
- GHS_EditorWindow Edit Functions
- GHS_EditorWindow File Functions
- GHS_EditorWindow Selection and Cursor Functions
- GHS_EditorWindow Version Control Functions

This chapter documents functions from the following classes:

- `GHS_Editor` (alias `ghs_editor`) — Exposes the basic functions of the MULTI Editor service. This class inherits from class `GHS_IdeObject`. See “GHS_Editor Functions” on page 266.
- `GHS_EditorWindow` — Exposes additional functions of the MULTI Editor. This class inherits from class `GHS_Window`.

The `GHS_EditorWindow` functions are divided into the following sections:

- “GHS_EditorWindow Edit Functions” on page 268
- “GHS_EditorWindow File Functions” on page 271
- “GHS_EditorWindow Selection and Cursor Functions” on page 273
- “GHS_EditorWindow Version Control Functions” on page 276

GHS_Editor Functions

The following sections describe functions from class `GHS_Editor`.

`__init__()`

```
__init__(serviceName="Editor.Multi", workingDir="")
```

Initializes the object attributes.

Arguments are:

- `serviceName` — Stores the MULTI IDE service name.
- `workingDir` — Stores the working directory of the MULTI service object.

EditFile()

```
EditFile(fileName, reuseEditorWindow=True,  
expandFileName=True)
```

Loads a file into a MULTI Editor window. On success, this function returns a GHS_EditorWindow object for the MULTI Editor window. On failure, it returns None.

Arguments are:

- `fileName` — Specifies the name of the file. If `fileName` is an empty string (""), the MULTI Editor opens a file chooser from which you can select a file.
- `reuseEditorWindow` — If `True`, reuses an existing MULTI Editor window. If `False`, does not reuse an existing window. If the file is already loaded into a MULTI Editor window, the file becomes the current file in the window.
- `expandFileName` — If `True`, uses the client's current working directory to expand `fileName` into a complete file path. If `False`, directly transfers `fileName` to the corresponding service, which resolves the filename with its current working directory if necessary. For more information, see the `expandFileName` description in “LoadWorkspaceFile()” on page 305.

Aliases are: `OpenFile()`, `Open()`

GotoLine()

```
GotoLine(fileName, lineNo, reuseEditorWindow=True)
```

Loads a file into a MULTI Editor window and moves the cursor to the specified line. On success, this function returns a GHS_EditorWindow object for the MULTI Editor window. On failure, it returns None.

Arguments are:

- `fileName` — Specifies the name of the file. If `fileName` is an empty string (""), the MULTI Editor opens a file chooser from which you can select a file.
- `lineNo` — Specifies the line number to go to.

- `reuseEditorWindow` — If `True`, reuses an existing `MULTI` Editor window. If `False`, does not reuse an existing window. If the file is already loaded into a `MULTI` Editor window, the file becomes the current file in the window.

The alias is: `Goto()`

GHS_EditorWindow Edit Functions

The following sections describe the edit functions from class `GHS_EditorWindow`.

AddString()

`AddString(dataStr="", pos=0)`

Adds a text string to the file that is open in the Editor. This function returns `True` on success and `False` on failure.

Arguments are:

- `dataStr` — Specifies the text string to add.
- `pos` — Indicates the position where the string is added. This argument may be:
 - `-1` — Adds the string to the beginning of the file.
 - `0` — Adds the string to the current cursor position. If there is a selection, the selection is replaced by the string.
 - `1` — Adds the string to the end of the file.

The alias is: `AddStr()`

Copy()

`Copy(which=1)`

Copies the selected string to the clipboard and returns `True` on success and `False` on failure.

The argument is:

- `which` — Indicates which clipboard to use. For more information, see “Clipboard Commands” in Appendix A, “Editor Commands” in the *MULTI: Editing Files and Configuring the IDE* book.

Cut()

`Cut (which=1)`

Cuts the selected string and stores it on the clipboard. This function returns `True` on success and `False` on failure.

The argument is:

- `which` — Indicates which clipboard to use. For more information, see “Clipboard Commands” in Appendix A, “Editor Commands” in the *MULTI: Editing Files and Configuring the IDE* book.

GetTextLines()

`GetTextLines (removeCrs=True)`

Returns a list of strings for the text in the Editor window. Each string in the list corresponds to a line in the Editor window. An empty list is returned upon error.

The argument is:

- `removeCrs` — If `True`, removes carriage return characters (if any). If `False`, does not remove carriage return characters.

Aliases are: `TextLines()`, `Lines()`

GetTextString()

`GetTextString()`

Returns a string for the text in the Editor window. An empty string ("") is returned upon error.

Aliases are: `TextString()`, `TextStr()`

Paste()

`Paste(which=1, waitingTime=1.0)`

Pastes the clipboard contents to the current cursor position. This function returns `True` on success and `False` on failure.

Arguments are:

- `which` — Indicates which clipboard to use. For more information, see “Clipboard Commands” in Appendix A, “Editor Commands” in the *MULTI: Editing Files and Configuring the IDE* book.
- `waitingTime` — Specifies the maximum number of seconds to wait for the paste operation to finish.

On UNIX, the MULTI Editor pastes a string in two parts:

1. It asks the X server to convert the selection.
2. It performs the paste on the X server’s selection notification.

The Editor’s **Paste** command returns immediately after the first part is done, even when the function is executed in blocked mode. Specifying a wait period allows the second part of the paste to finish.

Redo()

`Redo()`

Restores the edit that was reversed by `Undo()` (see “Undo()” on page 270). This function returns `True` on success and `False` on failure.

Undo()

`Undo()`

Reverses the last change made to the current file and returns `True` on success and `False` on failure.

GHS_EditorWindow File Functions

The following sections describe the file functions from class `GHS_EditorWindow`.

CloseCurrentFile()

`CloseCurrentFile()`

Closes the file currently displayed in the MULTI Editor window and returns `True` on success and `False` on failure.

Aliases are: `CloseCurFile()`, `CloseFile()`

GotoNextFile()

`GotoNextFile()`

Cycles to the next buffered file. The MULTI Editor keeps a circular list of loaded files, and you can navigate to each of them (that is, display the file in the MULTI Editor window). This function returns `True` on success and `False` on failure.

The alias is: `NextFile()`

GotoPrevFile()

`GotoPrevFile()`

Cycles to the previous buffered file. The MULTI Editor keeps a circular list of loaded files, and you can navigate to each of them (that is, display the file in the MULTI Editor window). This function returns `True` on success and `False` on failure.

The alias is: `PrevFile()`

OpenFile()

```
OpenFile(fileName="", newWindow=False, block=True)
```

Loads a file into a MULTI Editor window. On success, this function returns a `GHS_EditorWindow` object for the MULTI Editor window. On failure, it returns `None`.

Arguments are:

- `fileName` — Specifies the name of the file. If `fileName` is an empty string (`""`), the MULTI Editor opens a file chooser from which you can select a file.
- `newWindow` — If `True`, opens a new MULTI Editor window. If `False`, reuses an existing MULTI Editor window.
- `block` — If `True`, executes `OpenFile()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

Aliases are: `EditFile()`, `LoadFile()`, `Open()`

SaveAsFile()

```
SaveAsFile(fileName="")
```

Saves the current file (including changes) to another file. This function returns `True` on success and `False` on failure.

The argument is:

- `fileName` — Specifies the name of the file to save to. If `fileName` is an empty string (`""`), the MULTI Editor saves changes to the current file into the same file (that is, it performs a save rather than a save as).

The alias is: `SaveAs()`

SaveIntoFile()

`SaveIntoFile()`

Saves changes to the current file and returns `True` on success and `False` on failure.

Aliases are: `SaveFile()`, `Save()`

GHS_EditorWindow Selection and Cursor Functions

The following sections describe the selection and cursor-related functions from class `GHS_EditorWindow`.

FlashCursor()

`FlashCursor()`

Makes the location of the cursor obvious by flashing the line where it is located. This function returns `True` on success and `False` on failure.

The alias is: `Flash()`

GetSelection()

`GetSelection(printRes=False, getGuiPos=True)`

Gets the selection range and returns a tuple with the following components:

`(BeginLine, BeginColumn, EndLine, EndColumn)`

The following tuple is returned upon error:

`(-1, -1, -1, -1)`

Line numbering starts at 1. Column numbering starts at 1 by default.

Arguments are:

- `printRes` — If `True`, prints the selection range. If `False`, does not print the selection range.

- `getGuiPos` — If `True`, the `BeginColumn` and `EndColumn` values correspond to the one-based column positions displayed in the `MULTI` Editor status bar. A Tab is counted as the number of characters configured for the Tab width.

If `False`, the `BeginColumn` and `EndColumn` values are string indices, which are zero-based. A Tab is counted as one character.

The alias is: `GetSel()`

GetSelectedString()

`GetSelectedString(printRes=False)`

Gets the selected string. This function returns the string, or it returns an empty string (" ") upon failure.

The argument is:

- `printRes` — If `True`, prints the selected string. If `False`, does not print the selected string.

The alias is: `GetSelStr()`

MoveCursor()

`MoveCursor(line=1, col=1, flash=True)`

Moves the cursor to the specified position and returns `True` on success and `False` on failure.

Arguments are:

- `line` — Specifies the number of the line where you want to move the cursor. A 0 (zero) or a negative value indicates the end of the file.
- `col` — Specifies the number of the column where you want to move the cursor (the cursor comes before the specified column). A 0 (zero) or a negative value indicates the end of the line.

The alias is: `MoveTo()`

SelectAll()

`SelectAll()`

Selects all the content in the MULTI Editor window and returns `True` on success and `False` on failure.

The alias is: `SelAll()`

SelectRange()

`SelectRange(beginLine=-1, beginCol=-1, endLine=-1, endCol=-1, guiPos=True)`

Selects a range and returns `True` on success and `False` on failure.

Arguments are:

- `beginLine` — Specifies the number of the line where the selection begins. A 0 (zero) or a negative value indicates the first line.
- `beginCol` — Specifies the number of the column where the selection begins. A 0 (zero) or a negative value indicates the first column.
- `endLine` — Specifies the number of the line where the selection ends. A 0 (zero) or a negative value indicates the end of the file.
- `endCol` — Specifies the number of the column where the selection ends. A 0 (zero) or a negative value indicates the end of the line.
- `guiPos` — If `True`, interprets `beginCol` and `endCol` as corresponding to the one-based column positions displayed in the MULTI Editor status bar. A Tab is counted as the number of characters configured for the Tab width.

If `False`, `beginCol` and `endCol` are interpreted as string indices, which are zero-based. A Tab is counted as one character.

Aliases are: `SelRange()`, `Select()`, `Sel()`

GHS_EditorWindow Version Control Functions

The following sections describe the version control functions from class `GHS_EditorWindow`.

Checkin()

`Checkin()`

Checks the current file into version control and returns `True` on success and `False` on failure.

Checkout()

`Checkout()`

Checks the current file out of version control and returns `True` on success and `False` on failure.

PlaceUnderVC()

`PlaceUnderVC()`

Places the current file under version control and returns `True` on success and `False` on failure.

The alias is: `PlaceIntoVC()`

EventAnalyzer Functions

Chapter 14

This Chapter Contains:

- GHS_EventAnalyzer Functions
- GHS_EventAnalyzerWindow File Functions
- GHS_EventAnalyzerWindow View and Selection Functions
- GHS_EventAnalyzerWindow Miscellaneous Functions

This chapter documents functions from the following classes:

- `GHS_EventAnalyzer` (alias `ghs_eventanalyzer`) — Exposes the basic functions of the MULTI EventAnalyzer service. This class inherits from class `GHS_IdeObject`. See “GHS_EventAnalyzer Functions” on page 278.



Note When all MULTI EventAnalyzer windows opened on an instance of the class are closed, the instance of the service automatically shuts down, even if a Python object holds a reference to it.

- `GHS_EventAnalyzerWindow` — Exposes additional functions of the MULTI EventAnalyzer. This class inherits from class `GHS_Window`.

The `GHS_EventAnalyzerWindow` functions are divided into the following sections:

- “GHS_EventAnalyzerWindow File Functions” on page 281
- “GHS_EventAnalyzerWindow View and Selection Functions” on page 282
- “GHS_EventAnalyzerWindow Miscellaneous Functions” on page 286

For more information about the EventAnalyzer, see the *EventAnalyzer User’s Guide*.

GHS_EventAnalyzer Functions

The following sections describe functions from class `GHS_EventAnalyzer`.

`__init__()`

```
__init__(workingDir=" ")
```

Initializes the object attributes.

The argument is:

- `workingDir` — Stores the working directory of the MULTI service object.

CloseFile()

```
CloseFile(dataFile)
```

Closes the MULTI EventAnalyzer window displaying the specified event data file. This function returns `True` on success and `False` on failure.

The argument is:

- `dataFile` — Specifies the name of the event data file.

GetFileList()

```
GetFileList(showThem=True, onlyShowBaseName=False)
```

Gets a list of the data files displayed in MULTI EventAnalyzer windows. On success, this function returns a list of strings for the data files. On failure, it returns an empty list.

Arguments are:

- `showThem` — If `True`, displays the filenames. If `False`, does not display the filenames.
- `onlyShowBaseName` — If `True`, only displays the base names. If `False`, displays the full path to the filename.

Aliases are: `GetFiles()`, `Files()`

OpenFile()

```
OpenFile(dataFile="", title=None, newWin=True,  
raiseWin=True, reloadData=True)
```

Displays an event data file in the MULTI EventAnalyzer. On success, the function returns a `GHS_EventAnalyzerWindow` object for the MULTI EventAnalyzer window. On failure, it returns `None`.

Arguments are:

- `dataFile` — Specifies the name of the event data file.
- `title` — Specifies a title for the MULTI EventAnalyzer window.

- `newWin` — If `True`, displays the event data file in a new `MULTI EventAnalyzer` window. If `False`, reuses an existing `MULTI EventAnalyzer` window.
- `raiseWin` — If `True`, brings the reused `EventAnalyzer` to the foreground. If `False`, does not bring the `EventAnalyzer` to the foreground. This argument is only applicable if `newWin` is `False`.
- `reloadData` — If `True`, reloads the data file. If `False`, does not reload the data file. This argument is only applicable if `newWin` is `False`.

The alias is: `Open()`

ScrollToPosition()

```
ScrollToPosition(dataFile, timeValue,  
newWin=True, raiseWin=True, newWinIfNotExist=True,  
reloadData=True, oid=0)
```

Displays an event data file in the `MULTI EventAnalyzer` window and scrolls to the specified position. On success, this function returns a `GHS_EventAnalyzerWindow` object for the `MULTI EventAnalyzer` window. On failure, it returns `None`.

Arguments are:

- `dataFile` — Specifies the name of the event data file.
- `timeValue` — Specifies the time (in seconds or ticks) that you want to scroll to.
- `newWin` — If `True`, displays the event data file in a new `MULTI EventAnalyzer` window. If `False`, reuses an existing `MULTI EventAnalyzer` window.
- `raiseWin` — If `True`, brings the reused `EventAnalyzer` to the foreground. If `False`, does not bring the `EventAnalyzer` to the foreground. This argument is only applicable if `newWin` is `False`.
- `newWinIfNotExist` — If `True`, displays the event data file in a new `MULTI EventAnalyzer` window when the data file is not being shown in an existing `MULTI EventAnalyzer` window. If `False`, reuses an existing `MULTI EventAnalyzer` window.

- `reloadData` — If `True`, reloads the data file. If `False`, does not reload the data file. This argument is only applicable if `newWin` is `False`.
- `oid` — Specifies the ID of the task, thread, etc. that you want to scroll to.

Aliases are: `ScrollTo()`, `MoveTo()`

GHS_EventAnalyzerWindow File Functions

The following sections describe the file functions from class `GHS_EventAnalyzerWindow`.

CloseFile()

```
CloseFile(block=True, printOutput=True)
```

Closes the file displayed in the MULTI EventAnalyzer window and returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `CloseFile()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

OpenFile()

```
OpenFile(fileName="", block=True, printOutput=True)
```

Displays an event file in the MULTI EventAnalyzer window. This function returns `True` on success and `False` on failure.

Arguments are:

- `fileName` — Specifies the name of the file.
- `block` — If `True`, executes `OpenFile()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.

- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

The alias is: `LoadFile()`

GHS_EventAnalyzerWindow View and Selection Functions

The following sections describe the functions from class `GHS_EventAnalyzerWindow` that relate to viewing and selecting information in the `MULTI EventAnalyzer`.

GotoFirstView()

`GotoFirstView(block=True, printOutput=True)`

Displays the earliest view in the history and returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `GotoFirstView()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

The alias is: `FirstView()`

GotoLastView()

`GotoLastView(block=True, printOutput=True)`

Displays the latest view in the history and returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `GotoLastView()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.

- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

The alias is: `LastView()`

GotoNextView()

`GotoNextView(block=True, printOutput=True)`

Displays the next view in the history and returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `GotoNextView()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

The alias is: `NextView()`

GotoPrevView()

`GotoPrevView(block=True, printOutput=True)`

Displays the previous view in the history and returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `GotoPrevView()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

The alias is: `PrevView()`

SelectRange()

```
SelectRange(begin, end, unit="")
```

Selects a range in the MULTI EventAnalyzer window. This function returns `True` on success and `False` on failure.

Arguments are:

- `begin` — Specifies the range's starting time.
- `end` — Specifies the range's ending time.
- `unit` — Specifies the time unit used for the range. Valid `unit` values are:
 - `s` — seconds
 - `ms` — milliseconds
 - `us` — microseconds
 - `ns` — nanoseconds

If you do not specify a time unit, `begin` and `end` are assumed to be in seconds unless the event data does not include timestamps. In this case, ticks are used.

Aliases are: `SelRange()`, `Select()`, `Sel()`

ToggleFlatView()

```
ToggleFlatView(block=True, printOutput=True)
```

Toggles the object list between hierarchy and flat view. If the object list has no hierarchy, the operation has no effect. This function returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `ToggleFlatView()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

The alias is: `FlatView()`

ViewRange()

`ViewRange(begin, end, unit=" ")`

Allows you to view a range in the MULTI EventAnalyzer window. This function returns `True` on success and `False` on failure.

Arguments are:

- `begin` — Specifies the range's starting time.
- `end` — Specifies the range's ending time.
- `unit` — Specifies the time unit used for the range. For more information, see “`SelectRange()`” on page 284.

The alias is: `View()`

ZoomIn()

`ZoomIn()`

Zooms in on the MULTI EventAnalyzer window by a factor of 2. This function returns `True` on success and `False` on failure.

ZoomOut()

`ZoomOut()`

Zooms out from the MULTI EventAnalyzer window by a factor of 2. This function returns `True` on success and `False` on failure.

ZoomToSelection()

`ZoomToSelection()`

Zooms in on the current selection in the MULTI EventAnalyzer window. This function returns `True` on success and `False` on failure.

Aliases are: `ZoomToSel()`, `ZoomSel()`

GHS_EventAnalyzerWindow Miscellaneous Functions

The following sections describe the miscellaneous functions from class `GHS_EventAnalyzerWindow`.

AutoTimeUnit()

```
AutoTimeUnit (toggle=True, auto=True)
```

Changes how the time unit is determined. This function returns `True` on success and `False` on failure.

Arguments are:

- `toggle` — If `True`, toggles between automatically adjusting the time unit and requiring you to manually change the time unit. If `False`, does not toggle these settings.
- `auto` — If `True`, the EventAnalyzer automatically adjusts to an appropriate time unit whenever the view is changed. If `False`, you must manually set the time unit. This argument is only effective when `toggle` is `False`.

The alias is: `AutoUnit()`

ChangeTimeUnit()

```
ChangeTimeUnit (unit="", gui=False)
```

Changes the time unit. This function returns `True` on success and `False` on failure.

Arguments are:

- `unit` — Specifies the time unit of the range. If you do not specify a time unit, `unit` is set to the next larger time unit. For more information, see “`SelectRange()`” on page 284.
- `gui` — If `True`, displays a dialog box from which you can choose a time unit. If `False`, does not display a dialog box.

Aliases are: `ChangeUnit()`, `Unit()`

NewWindow()

`NewWindow()`

Opens a new MULTI EventAnalyzer window. On success, this function returns a `GHS_EventAnalyzerWindow` object for the MULTI EventAnalyzer window. On failure, it returns `None`.

The alias is: `NewWin()`

SaveMevConfiguration()

`SaveMevConfiguration(block=True, printOutput=True)`

Saves configuration changes to the configuration file. This function returns `True` on success and `False` on failure.

Arguments are:

- `block` — If `True`, executes `SaveMevConfiguration()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

The alias is: `SaveMevConfig()`

ShowLegend()

`ShowLegend()`

Displays the **Legend** window and returns `True` on success and `False` on failure.

The alias is: `Legend()`

Launcher Functions

Chapter 15

This Chapter Contains:

- GHS_Action Attributes and Functions
- GHS_ActionSequence Attributes and Functions
- GHS_Variable Attributes and Functions
- GHS_Workspace Attributes and Functions
- GHS_Launcher Functions
- GHS_LauncherWindow Action Execution Functions
- GHS_LauncherWindow Action Manipulation Functions
- GHS_LauncherWindow Workspace Manipulation Functions
- GHS_LauncherWindow Variable Functions

This chapter documents functions from the following utility classes:

- `GHS_Action` — Stores information for an action. This class inherits from `object` and is a utility class for `GHS_LauncherWindow`. See “GHS_Action Attributes and Functions” on page 291.
- `GHS_ActionSequence` — Stores information for an action sequence. This class inherits from `object` and is a utility class for `GHS_LauncherWindow`. See “GHS_ActionSequence Attributes and Functions” on page 292.
- `GHS_Variable` — Stores information for a Launcher variable. This class inherits from `object` and is a utility class for `GHS_LauncherWindow`. See “GHS_Variable Attributes and Functions” on page 293.
- `GHS_Workspace` — Stores information for a workspace. This class inherits from `object` and is a utility class for `GHS_LauncherWindow`. See “GHS_Workspace Attributes and Functions” on page 294.

This chapter also covers functions from classes:

- `GHS_Launcher` (alias `ghs_launcher`) — Exposes the basic functions of the MULTI Launcher service. This class inherits from class `GHS_LauncherWindow` because the MULTI Launcher service is always associated with a GUI window. See “GHS_Launcher Functions” on page 297.
- `GHS_LauncherWindow` — Exposes the functions of the MULTI Launcher. This class inherits from class `GHS_Window`. The `GHS_LauncherWindow` functions are divided into the following sections:
 - “GHS_LauncherWindow Action Execution Functions” on page 298
 - “GHS_LauncherWindow Action Manipulation Functions” on page 301
 - “GHS_LauncherWindow Workspace Manipulation Functions” on page 303
 - “GHS_LauncherWindow Variable Functions” on page 307

GHS_Action Attributes and Functions

Class `GHS_Action` stores information for an action.

The following list describes the attributes of this class:

- `name` — Stores the action type (for example, `Editor` or `Python Statement`). For more action types, see “Creating or Modifying an Action” in Chapter 5, “Using MULTI Workspaces and Shortcuts” in the *MULTI: Editing Files and Configuring the IDE* book.
- `args` — Stores action arguments. The validity of an argument depends on the value of `name`. For example, an `Editor` action may take a file path or a filename as an argument, while a `Python Statement` may accept a Python statement as an argument. For more information, see “Creating or Modifying an Action” in Chapter 5, “Using MULTI Workspaces and Shortcuts” in the *MULTI: Editing Files and Configuring the IDE* book.
- `enabled` — Stores the status of the action (that is, whether it’s enabled or disabled). Disabled actions do not run when the action sequence containing them is executed.
- `parent` — Stores the `GHS_ActionSequence` object that contains the action object.

The following section describes the function from class `GHS_Action`.

DumpTree()

`DumpTree(treeLine=False)`

Dumps the action type, action arguments, and whether the action is enabled or disabled. This function returns `True` on success.

The argument is:

- `treeLine` — If `True`, lines such as `|` and `_` are printed to show the hierarchy of the dumped information. For example, one of the three lines below, beginning with `|`:

```
Action Sequence
| Editor      test.cc
| Editor      test1.cc
|_Python Statement    x = 1;
```

If `False`, the lines are not printed.

The alias is: `Dump()`

GHS_ActionSequence Attributes and Functions

Class `GHS_ActionSequence` stores information for an action sequence.

The following list describes the attributes of this class:

- `name` — Stores the name of the action sequence.
- `actions` — Stores a list of `GHS_Action` objects.
- `parent` — Stores the `GHS_Workspace` object that contains the action sequence object.

The following sections describe functions from class `GHS_ActionSequence`.

DumpTree()

`DumpTree(treeLine=True)`

Dumps the action sequence's name and actions and returns `True` on success.

The argument is:

- `treeLine` — If `True`, lines such as `|` and `_` are printed to show the hierarchy of the dumped information. If `False`, the lines are not printed. For an example, see “`DumpTree()`” on page 291.

The alias is: `Dump()`

Search()

`Search(value, column=-1, match=False, all=False)`

Searches for one or more actions in an action sequence.

Arguments are:

- `value` — Specifies the value (action name or action argument) to search for. This can be a regular expression.
- `column` — Specifies whether action names, action arguments, or both names and arguments are searched. If 0, names are searched. If 1, arguments are searched. If -1, both names and arguments are searched.
- `match` — If `True`, uses Python's regular expression `match()` function to check the specified value. If `False`, uses Python's regular expression `search()` function.
- `all` — If `True`, a list of action objects satisfying the criteria is returned upon success. Upon failure, an empty list is returned. If `False`, the first action object satisfying the criteria is returned upon success. Upon failure, `None` is returned.

GHS_Variable Attributes and Functions

Class `GHS_Variable` stores information for a Launcher variable.

The following list describes the attributes of this class:

- `name` — Stores the name of the variable.
- `value` — Stores the value of the variable.
- `type` — Stores the variable type. The type may be `local`, `global`, or `predefined`. For more information, see “Variable Types” in Chapter 5, “Using MULTI Workspaces and Shortcuts” in the *MULTI: Editing Files and Configuring the IDE* book.
- `parent` — Stores the `GHS_Workspace` object that contains the variable.

The following section describes the function from class `GHS_Variable`.

DumpTree()

`DumpTree(treeLine=True)`

Dumps the variable's name and value and returns `True` on success.

The argument is:

- `treeLine` — If `True`, lines such as `|` and `_` are printed to show the hierarchy of the dumped information. If `False`, the lines are not printed. For an example, see “`DumpTree()`” on page 291.

The alias is: `Dump()`

GHS_Workspace Attributes and Functions

Class `GHS_Workspace` stores information for a workspace.

The following list describes the attributes of this class:

- `name` — Stores the name of the workspace.
- `workingDir` — Stores the workspace’s working directory. The working directory is the directory from which action sequences are started.
- `localVariables` — Stores a list of variables local to the workspace.
- `globalVariables` — Stores a list of variables available to all workspaces.
- `predefinedVariables` — Stores a list of pre-defined Launcher variables. For more information, see “Variable Types” in Chapter 5, “Using MULTI Workspaces and Shortcuts” in the *MULTI: Editing Files and Configuring the IDE* book.
- `actionSequences` — Stores a list of the action sequences contained in the workspace.
- `dumpNameMask` (for constant `0x1`) — A flag for dumping the workspace name.
- `dumpWorkingDirMask` (for constant `0x2`) — A flag for dumping the working directory of the workspace.
- `dumpLocalVarsMask` (for constant `0x4`) — A flag for dumping the workspace’s local variables.
- `dumpGlobalVarsMask` (for constant `0x8`) — A flag for dumping the global variables.
- `dumpPredefinedVarsMask` (for constant `0x10`) — A flag for dumping the pre-defined MULTI Launcher variables.

- `dumpActionsMask` (for constant `0x20`) — A flag for dumping the workspace's actions.

The following sections describe functions from class `GHS_Workspace`.

DumpTree()

```
DumpTree(treeLine=True, mask=0xff)
```

Dumps the workspace's name, working directory, local variables, global variables, pre-defined MULTI Launcher variables, action sequences, and actions; or dumps whichever of these elements you specify. This function returns `True` on success.

Arguments are:

- `treeLine` — If `True`, lines such as `|` and `_` are printed to show the hierarchy of the dumped information. If `False`, the lines are not printed. For an example, see “`DumpTree()`” on page 291.
- `mask` — Controls what elements are dumped. You can specify any combination of the `dump*` flags listed in “GHS_Workspace Attributes and Functions” on page 294. If you specify multiple flags, use the OR operation to link them together. By default, this function dumps all elements of the workspace.

The alias is: `Dump()`

SearchAction()

```
SearchAction(asName, value, column=-1, match=False, all=False)
```

Searches for one or more actions in the workspace.

Arguments are:

- `asName` — Specifies the name of the action sequence to search. This can be a regular expression. If `asName` is an empty string (`""`), all action sequences are searched.
- `value` — Specifies the value (action name or action argument) to search for. This can be a regular expression.

- `column` — Specifies whether action names, action arguments, or both names and arguments are searched. If 0, names are searched. If 1, arguments are searched. If -1, names and arguments are searched.
- `match` — If `True`, uses Python's regular expression `match()` function to check the specified action. If `False`, uses Python's regular expression `search()` function.
- `all` — If `True`, a list of action objects satisfying the criteria is returned upon success. Upon failure, an empty list is returned. If `False`, the first action satisfying the criteria is returned upon success. Upon failure, `None` is returned.

SearchActionSequence()

```
SearchActionSequence(asName, match=False, all=False)
```

Searches for one or more action sequences in a workspace.

Arguments are:

- `asName` — Specifies the name of the action sequence. This can be a regular expression.
- `match` — If `True`, uses Python's regular expression `match()` function to check the specified action sequence name. If `False`, uses Python's regular expression `search()` function.
- `all` — If `True`, a list of action sequence objects satisfying the criteria is returned upon success. Upon failure, an empty list is returned. If `False`, the first action object satisfying the criteria is returned upon success. Upon failure, `None` is returned.

The alias is: `SearchActionSeq()`

SearchVariable()

```
SearchVariable(varType, value, column=-1,  
match=False, all=False)
```

Searches for one or more variables in the workspace.

Arguments are:

- `varType` — Specifies the variable type. The `varType` may be:
 - `local` — Specifies that the variable is local to the workspace.
 - `global` — Specifies that the variable is available to all workspaces.
 - `predefined` — Specifies that the variable is a pre-defined Launcher variable. For more information, see “Variable Types” in Chapter 5, “Using MULTI Workspaces and Shortcuts” in the *MULTI: Editing Files and Configuring the IDE* book.
 - `" "` — Specifies all variable types.
- `value` — Specifies the value (variable name or variable value) to search for. This can be a regular expression.
- `column` — Specifies whether variable names, variable values, or both names and values are searched. If 0, names are searched. If 1, values are searched. If -1, names and values are searched.
- `match` — If `True`, uses Python’s regular expression `match()` function to check the specified variable. If `False`, uses Python’s regular expression `search()` function.
- `all` — If `True`, a list of variable objects satisfying the criteria is returned upon success. Upon failure, an empty list is returned. If `False`, the first variable satisfying the criteria is returned upon success. Upon failure, `None` is returned.

The alias is: `SearchVar()`

GHS_Launcher Functions

The following section describes the `__init__()` function from class `GHS_Launcher`.

`__init__()`

```
__init__(workingDir=" ")
```

Initializes a `GHS_Launcher` object.

This function also gets the window's information for the created MULTI Launcher service and sets up the object with it.

The argument is:

- `workingDir` — Specifies the working directory of the MULTI IDE service.

GHS_LauncherWindow Action Execution Functions

The following sections describe the functions from class `GHS_LauncherWindow` that relate to action execution.

GetRunningActions()

```
GetRunningActions (show=True)
```

Gets a list of actions that are currently running. This function returns a list of strings, where each string corresponds to a running action.

The argument is:

- `show` — If `True`, displays the running actions (if any). If `False`, does not display running actions.

RunAction()

```
RunAction(actionType, actionArgs="", workingDir="",  
wsName="", waitPeriodToFinish=0.0)
```

Executes the specified ad hoc action. This function returns `True` on success and `False` on failure.

Arguments are:

- `actionType` — Specifies the type of the action to be executed. For a list of supported action types, see “Creating or Modifying an Action” in Chapter 5, “Using MULTI Workspaces and Shortcuts” in the *MULTI: Editing Files and Configuring the IDE* book.
- `actionArgs` — Specifies the argument(s) of the action.

- `workingDir` — Specifies the working directory from which to execute the action.
- `wsName` — Specifies the name of the workspace whose working directory is used to execute the action.
- `waitPeriodToFinish` — Specifies how long the function waits for the action(s) to finish, if it waits at all. This argument may be:
 - 0 or a negative number — Indicates that the function does not wait.
 - A positive number — Specifies the maximum number of seconds that the function waits. If the action has not finished before the timeout, the function returns `False`.

If `workingDir` is specified (that is, it is not an empty string), the action is executed from the directory specified. Otherwise, if `wsName` is specified (that is, it is not an empty string) and the workspace name exists, the action is executed from the corresponding workspace's working directory. If neither `workingDir` nor `wsName` is specified, the action executes from the working directory of the Launcher's current workspace.

RunWorkspaceAction()

```
RunWorkspaceAction(workspaceName="",  
actionSequenceName="", actionIndex=-1,  
waitPeriodToFinish=0.0)
```

Executes one or more actions in a workspace action sequence and returns `True` on success and `False` on failure.

Arguments are:

- `workspaceName` — Specifies the name of the workspace whose actions are executed. If no workspace name is given, the Launcher's current workspace is used.
- `actionSequenceName` — Specifies the name of the action sequence whose actions are executed. If `actionSequenceName` is an empty string (`" "`), all enabled actions in the workspace are executed.
- `actionIndex` — Specifies the action's index in the action sequence. The index starts at 0 (zero). If `actionIndex` is negative number, all enabled actions in the corresponding action sequence are executed.

- `waitPeriodToFinish` — Specifies how long the function waits for the action(s) to finish, if it waits at all. This argument may be:
 - 0 or a negative number — Indicates that the function does not wait.
 - A positive number — Specifies the maximum number of seconds that the function waits. If the action has not finished before the timeout, the function returns `False`.

The alias is: `RunWsAction()`

WaitForActionsToFinish()

```
WaitForActionsToFinish(oldActionList, duration=0.0,  
warnIfNotFinish=False)
```

Waits for running actions to finish. This function returns `True` if the running actions are finished when the function returns, and `False` otherwise.

Arguments are:

- `oldActionList` — Specifies an action list whose actions are not to be considered (that is, the function can return `True` even if actions in this list are still running). The function waits for all running actions (not in the old action list) to finish or timeout. If this argument is `None` or empty, the function waits for all running actions to finish or timeout.
- `duration` — Specifies how long the function waits, if at all. The `duration` may be:
 - A negative number — Indicates that the function waits until the running actions finish.
 - 0.0 — Indicates that the function does not wait.
 - A positive number — Specifies the maximum number of seconds that the function waits.

During the wait, the function checks the running actions every half second.

- `warnIfNotFinish` — If `True` and if the function returns when actions are still running, prints a warning message. If `False`, does not print a warning message.

Aliases are: `WaitToFinish()`, `Wait()`

GHS_LauncherWindow Action Manipulation Functions

The following sections describe the functions from class `GHS_LauncherWindow` that relate to action manipulation.

AddAction()

```
AddAction(actionType, actionArgs="",  
actionSequenceName="Startup",  
indexInActionSequence=-1, wsName="",  
workingDir="", block=True, printOutput=True)
```

Adds an action into a workspace and returns `True` on success and `False` on failure.

Arguments are:

- `actionType` — Specifies the action type. For the list of supported action types, see “Creating or Modifying an Action” in Chapter 5, “Using MULTI Workspaces and Shortcuts” in the *MULTI: Editing Files and Configuring the IDE* book.
- `actionArgs` — Specifies the argument(s) of the action.
- `actionSequenceName` — Specifies the name of the action sequence to which the action belongs.
- `indexInActionSequence` — Specifies the index of the position in the action sequence where the action is to be inserted. The index starts at 0 (zero). If you specify a negative value, the action is given the last position in the action sequence.
- `wsName` — Specifies the name of the workspace to which the action belongs.
- `workingDir` — Specifies the working directory of the workspace. This argument is used only when the specified workspace name does not exist (a new workspace is created in this case).
- `block` — If `True`, executes `AddAction()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

DeleteAction()

```
DeleteAction(actionSequenceName="",  
indexInActionSequence=-1, wsName="", block=True,  
printOutput=True)
```

Deletes the action(s) located at the specified position(s) from the given workspace action sequence. This function returns `True` on success and `False` on failure.

Arguments are:

- `actionSequenceName` — Specifies the name of the action sequence to which the action belongs. If `actionSequenceName` is an empty string (""), all actions in the workspace are deleted.
- `indexInActionSequence` — Specifies the action's index in the action sequence. The index starts at 0 (zero). If you specify a negative value, the entire action sequence is deleted.
- `wsName` — Specifies the name of the workspace to which the action belongs. If the workspace name is an empty string (""), the deletion is applied to the MULTI Launcher's current workspace.
- `block` — If `True`, executes `DeleteAction()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

The alias is: `DelAction()`

GHS_LauncherWindow Workspace Manipulation Functions

The following sections describe the functions from class `GHS_LauncherWindow` that relate to workspace manipulation.

CreateWorkspace()

```
CreateWorkspace(wsName="", workingDir="", block=True,
printOutput=True)
```

Creates a workspace and selects it as the current workspace. This function returns `True` on success and `False` on failure.

Arguments are:

- `wsName` — Specifies the name of the workspace.
- `workingDir` — Specifies the working directory of the workspace.
- `block` — If `True`, executes `CreateWorkspace()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

If both the workspace name and working directory are specified, the function creates an empty workspace with the given information. If the workspace name already exists, the workspace is selected into the current workspace, and the working directory (if it is specified) is applied to the existing workspace.

Aliases are: `CreateWs()`, `AddWorkspace()`, `AddWs()`

DeleteWorkspace()

```
DeleteWorkspace(wsName="", delCurrentWs=True,
block=True, printOutput=True)
```

Deletes the specified workspace and returns `True` on success and `False` on failure.

Arguments are:

- `wsName` — Specifies the name of the workspace.
- `delCurrentWs` — If `True`, deletes the current workspace. If `False`, does not delete the current workspace. Deleting the current workspace is only effective when `wsName` is an empty string (`" "`).
- `block` — If `True`, executes `DeleteWorkspace()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

The alias is: `DelWs()`

GetWorkspaceInformation()

```
GetWorkspaceInformation(wsName=" ", original=True)
```

Gets a workspace's name, working directory, local variables, global variables, pre-defined MULTI Launcher variables, and actions. This function returns a `GHS_Workspace` object (a read-only workspace tree) on success, or it returns `None` upon failure.

Arguments are:

- `wsName` — Specifies the name of the workspace. If `wsName` is an empty string (`" "`), this function gets information for the workspace currently displayed in the MULTI Launcher.
- `original` — If `True`, the variable values and action arguments are kept in the same format as in the MULTI Launcher workspace. If `False`, this function substitutes actual values for the variables nested in variable values and action arguments.

Aliases are: `GetWorkspaceInfo()`, `GetWsInfo()`

GetWorkspaces()

```
GetWorkspaces()
```

Gets a list of workspace names and returns the list.

The alias is: `GetWses()`

LoadWorkspaceFile()

```
LoadWorkspaceFile(fileName, block=True,  
printOutput=True, expandFileName=True)
```

Loads a workspace file into the MULTI Launcher and selects the corresponding workspace as the current workspace. This function returns `True` on success and `False` on failure.

Arguments are:

- `fileName` — Specifies the filename of the workspace to be loaded.
- `block` — If `True`, executes `LoadWorkspaceFile()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.
- `expandFileName` — If `True`, uses the client's current working directory to expand `fileName` into a complete file path. If `False`, directly transfers `fileName` to the MULTI Launcher service, which resolves the filename with its current working directory if necessary.



Note In the MULTI-Python integrated system, filenames are passed by clients to MULTI services, which may expand them. Example clients are: the **mpythonrun** utility program, the stand-alone Python GUI, and the MULTI Debugger (the Debugger contains a **Py** pane and is capable of running Python statements via buttons, menu items, etc.). For more information about MULTI services or about clients from which you can execute Python statements, see Chapter 3, “Introduction to the MULTI-Python Integration”.

Aliases are: `LoadWorkspace()`, `LoadWsFile()`, `LoadWs()`

SaveWorkspaceIntoFile()

```
SaveWorkspaceIntoFile(fileName="", wsName="",  
block=True, printOutput=True, expandFileName=True)
```

Saves a workspace into a file and returns `True` on success and `False` on failure.

Arguments are:

- `fileName` — Specifies the filename of the workspace to be saved. If no filename is specified, the MULTI Launcher opens a file chooser from which you can select a filename.
- `wsName` — Specifies the name of the workspace to be saved. If `wsName` is an empty string (`""`), the current workspace is saved.
- `block` — If `True`, executes `SaveWorkspaceIntoFile()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.
- `expandFileName` — If `True`, uses the client's current working directory to expand `fileName` into a complete file path. If `False`, directly transfers `fileName` to the MULTI Launcher service, which resolves the filename with its current working directory if necessary. For more information, see the `expandFileName` description in “`LoadWorkspaceFile()`” on page 305.

Aliases are: `SaveWsIntoFile()`, `SaveWs()`

SelectWorkspace()

```
SelectWorkspace(wsName, block=True, printOutput=True)
```

Selects the specified workspace as the current workspace. This function returns `True` on success and `False` on failure.

Arguments are:

- `wsName` — Specifies the name of the workspace.

- `block` — If `True`, executes `SelectWorkspace()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

The alias is: `SelWs()`

GHS_LauncherWindow Variable Functions

The following sections describe the functions from class `GHS_LauncherWindow` that relate to variables.

AddVariable()

```
AddVariable(varName, varValue, globalVar=False,  
wsName="", block=True, printOutput=True)
```

Adds a local variable to a workspace or a global variable to the system. If the variable already exists, its existing value is changed to the given value. This function returns `True` on success and `False` on failure.

Arguments are:

- `varName` — Specifies the name of the variable.
- `varValue` — Specifies the value of the variable.
- `globalVar` — If `True`, specifies that the variable is global. If `False`, specifies that it's local to a workspace.
- `wsName` — Specifies the name of the workspace to add the variable to. If `wsName` is an empty string (`" "`), this function adds the variable to the workspace currently displayed in the MULTI Launcher. This argument is only applicable if `globalVar` is `False`.
- `block` — If `True`, executes `AddVariable()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

The alias is: `AddVar()`

ChangeVariable()

```
ChangeVariable(varName, varValue, globalVar=False,  
wsName="", block=True, printOutput=True)
```

Changes the value of a local workspace variable or a global system variable. If the variable does not exist, it is added to the workspace or to the system. This function returns `True` on success and `False` on failure.

Arguments are:

- `varName` — Specifies the name of the variable.
- `varValue` — Specifies the new value of the variable.
- `globalVar` — If `True`, specifies that the variable is global. If `False`, specifies that it's local to a workspace.
- `wsName` — Specifies the name of the workspace that the variable is local to. If `wsName` is an empty string (`"`), this function changes the specified variable belonging to the workspace currently displayed in the **MULTI Launcher**. This argument is only applicable if `globalVar` is `False`.
- `block` — If `True`, executes `ChangeVariable()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

The alias is: `ChangeVar()`

DeleteVariable()

```
DeleteVariable(varName, globalVar=False, wsName="",  
block=True, printOutput=True)
```

Deletes a local variable from a workspace or a global variable from the system. This function returns `True` on success and `False` on failure.

Arguments are:

- `varName` — Specifies the name of the variable.
- `globalVar` — If `True`, specifies that the variable is global. If `False`, specifies that it's local to a workspace.
- `wsName` — Specifies the name of the workspace to delete the variable from. If `wsName` is an empty string (`" "`), this function deletes the specified variable from the workspace currently displayed in the MULTI Launcher. This argument is only applicable if `globalVar` is `False`.
- `block` — If `True`, executes `DeleteVariable()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print the output.

The alias is: `DelVar()`

Project Manager Functions

Chapter 16

This Chapter Contains:

- GHS_ProjectManager Functions
- GHS_ProjectManagerWindow Build Functions
- GHS_ProjectManagerWindow Edit Functions
- GHS_ProjectManagerWindow File Functions
- GHS_ProjectManagerWindow Navigation Functions
- GHS_ProjectManagerWindow Debug and Edit Functions
- GHS_ProjectManagerWindow Tree Expansion/Contraction Functions
- GHS_ProjectManagerWindow Selection Functions

This chapter documents functions from the following classes:

- `GHS_ProjectManager` (alias `ghs_projectmanager`) — Exposes the basic functions of the MULTI Project Manager service. This class inherits from class `GHS_IdeObject`. See “GHS_ProjectManager Functions” on page 312.
- `GHS_ProjectManagerWindow` — Exposes additional functions of the MULTI Project Manager. This class inherits from class `GHS_Window`.

The `GHS_ProjectManagerWindow` functions are divided into the following sections:

- “GHS_ProjectManagerWindow Build Functions” on page 314
- “GHS_ProjectManagerWindow Edit Functions” on page 317
- “GHS_ProjectManagerWindow File Functions” on page 318
- “GHS_ProjectManagerWindow Navigation Functions” on page 321
- “GHS_ProjectManagerWindow Debug and Edit Functions” on page 322
- “GHS_ProjectManagerWindow Tree Expansion/Contraction Functions” on page 323
- “GHS_ProjectManagerWindow Selection Functions” on page 325

GHS_ProjectManager Functions

The following sections describe functions from class `GHS_ProjectManager`.

`__init__()`

`__init__(workingDir="")`

Initializes the object attributes.

The argument is:

- `workingDir` — Stores the working directory of the MULTI service object.

GetTopProjectFiles()

```
GetTopProjectFiles (showList=False)
```

Gets top project files loaded in MULTI Project Manager windows and returns a list of the top project files.

The argument is:

- `showList` — If `True`, displays the top project files. If `False`, does not display them.

The alias is: `TopProjs()`

OpenProject()

```
OpenProject (fileName, targetsToBuild="",  
hidden=False, expandFileName=True)
```

Loads a MULTI project file into the MULTI Project Manager. On success, this function returns a `GHS_ProjectManagerWindow` object for the MULTI Project Manager window. Otherwise, it returns `None`.

Arguments are:

- `fileName` — Specifies the name of the project file. If the `fileName` is an empty string (`" "`), the launched MULTI Project Manager window could be blank.
- `targetsToBuild` — Specifies a list of targets to be built.
- `hidden` — If `True`, hides the MULTI Project Manager window. If `False`, does not hide the window.
- `expandFileName` — If `True`, uses the client's current working directory to expand `fileName` into a complete file path. If `False`, directly transfers `fileName` to the corresponding service, which resolves the filename with its current working directory if necessary. For more information, see the `expandFileName` description in “`LoadWorkspaceFile()`” on page 305.

Aliases are: `OpenFile()`, `OpenProj()`, `LoadFile()`, `Open()`

GHS_ProjectManagerWindow Build Functions

The following sections describe the build functions from class GHS_ProjectManagerWindow.

BuildAllProjects()

```
BuildAllProjects(waitBuildFinish=False,  
duration=-1.0)
```

Builds all projects and returns `True` on success and `False` on failure.

Arguments are:

- `waitBuildFinish` — If `True`, waits for the build to finish. If `False`, does not wait.
- `duration` — Specifies how long the function waits, if at all. The duration may be:
 - A negative number — Indicates that the function waits until the build is done.
 - 0 — Indicates that the function does not wait at all.
 - A positive number — Specifies the maximum number of seconds that the function waits. If the build is not finished within the given period, the function returns `False`, as with function failure.

During the wait, the function checks the status of the build approximately every 0.3 seconds.

Aliases are: `BuildAll()`, `BuildAllProj()`, `BuildAllProjs()`

BuildFile()

```
BuildFile(optionsAndFileName, block=True,  
printOutput=True, duration=-1)
```

Builds one or more of the files or projects in the Project Manager and returns `True` on success and `False` on failure.

Arguments are:

- `optionsAndFileName` — Specifies the names of the files and/or projects to build and, optionally, **gbuild** options to run. Specify this information in the same way that you would if you were entering it in the Project Manager's file shortcut bar (with **Build** selected). For more information, see "Using the File Shortcut Bar" in Chapter 2, "The MULTI Project Manager" in the *MULTI: Building Applications* book.
- `block` — If `True`, executes `BuildFile()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.
- `duration` — Specifies how long the function waits, if at all. For more information, see the `duration` description in "BuildAllProjects()" on page 314.

The alias is: `Build()`

BuildProjects()

```
BuildProjects(buildAll=False, waitBuildFinish=False,  
duration=-1.0)
```

Builds all projects or builds the selected projects. This function returns `True` on success and `False` on failure.

Arguments are:

- `buildAll` — If `True`, builds all projects. If `False`, builds selected projects.
- `waitBuildFinish` — If `True`, waits for the build to finish. If `False`, does not wait.
- `duration` — Specifies how long the function waits, if at all. For more information, see the `duration` description in "BuildAllProjects()" on page 314.

Aliases are: `BuildProjs()`, `BuildProj()`

BuildSelectedProjects()

```
BuildSelectedProjects(waitBuildFinish=False,  
duration=-1.0)
```

Builds selected projects and returns `True` on success and `False` on failure.

For argument descriptions, see “BuildAllProjects()” on page 314.

Aliases are: `BuildSelected()`, `BuildSelectedProjs()`,
`BuildSelectedProj()`, `BuildSel()`

GetStatus()

```
GetStatus()
```

Gets the status of the build. On success, this function returns a string with the status. Upon failure, it returns `None`.

HaltBuild()

```
HaltBuild(waitHaltFinish=False, duration=-1.0)
```

Halts the build and returns `True` on success and `False` on failure.

Arguments are:

- `waitHaltFinish` — If `True`, waits for the build to halt. If `False`, does not wait.
- `duration` — Specifies how long the function waits, if at all. The `duration` may be:
 - A negative number — Indicates that the function waits until the build is halted.
 - 0 — Indicates that the function does not wait at all.
 - A positive number — Specifies the maximum number of seconds that the function waits. If the build has not halted within the given period, the function returns `False`, as with function failure.

During the wait, the function checks the status of the build approximately every 0.3 seconds.

The alias is: `Halt()`

WaitForBuildingFinish()

`WaitForBuildingFinish(duration=-1.0, checkInterval=0)`

Waits for the build to finish and returns `True` on success and `False` on failure.

Arguments are:

- `duration` — Specifies how long the function waits, if at all. For more information, see the `duration` description in “`BuildAllProjects()`” on page 314.
- `checkInterval` — Specifies the interval (in seconds) between status checks. A 0 or a negative value specifies that the default check interval value of the MULTI Project Manager window object is used. The default check interval value (attribute `checkInterval`) for the MULTI Project Manager window object is 0.3.

GHS_ProjectManagerWindow Edit Functions

The following sections describe the edit functions from class `GHS_ProjectManagerWindow`.

CopySelected()

`CopySelected()`

Copies the selected project tree entries to a clipboard specific to the MULTI Project Manager. This function returns `True` on success and `False` on failure.

The alias is: `CopySel()`

CutSelected()

`CutSelected()`

Cuts the selected project tree entries and stores them on a clipboard specific to the MULTI Project Manager. This function returns `True` on success and `False` on failure.

The alias is: `CutSel()`

DeleteSelected()

`DeleteSelected()`

Deletes the selected project tree entries. This function returns `True` on success and `False` on failure.

Aliases are: `DelSelected()`, `DelSel()`

PasteAfterSelected()

`PasteAfterSelected()`

Pastes entries stored on the MULTI Project Manager clipboard after the selected entries. This function returns `True` on success and `False` on failure.

The alias is: `Paste()`

GHS_ProjectManagerWindow File Functions

The following sections describe the functions from class `GHS_ProjectManagerWindow` that relate to project files.

CloseProject()

`CloseProject()`

Closes the current project file in the MULTI Project Manager window. This function returns `True` on success and `False` on failure.

The alias is: `CloseProj()`

NewWindow()

`NewWindow()`

Opens a new MULTI Project Manager window. On success, this function returns a `GHS_ProjectManagerWindow` object for the new MULTI Project Manager window. On failure, it returns `None`.

The alias is: `NewWin()`

OpenProject()

`OpenProject(projName="", block=True, expandFileName=True)`

Loads a project file into the MULTI Project Manager window and returns `True` on success and `False` on failure.

Arguments are:

- `projName` — Specifies the name of the project file. If `projName` is an empty string (`""`), the MULTI Project Manager opens a file chooser from which you can select a project file.
- `block` — If `True`, executes `OpenProject()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.
- `expandFileName` — If `True`, uses the client's current working directory to expand `projName` into a complete file path. If `False`, directly transfers `projName` to the corresponding service, which resolves the filename with its current working directory if necessary. For more information, see the `expandFileName` description in “`LoadWorkspaceFile()`” on page 305.

Aliases are: `OpenFile()`, `OpenProj()`, `LoadFile()`, `Open()`

RevertFromFile()

`RevertFromFile (block=True)`

Reverts the contents of the MULTI Project Manager window to the corresponding file saved on disk. This function returns `True` on success and `False` on failure.

The argument is:

- `block` — If `True`, executes `RevertFromFile()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

The alias is: `Revert()`

SaveChanges()

`SaveChanges (block=True)`

Saves changes made in the MULTI Project Manager window to the corresponding project file. This function returns `True` on success and `False` on failure.

The argument is:

- `block` — If `True`, executes `SaveChanges()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

Aliases are: `SaveChange()`, `Save()`

GHS_ProjectManagerWindow Navigation Functions

The following sections describe the file navigation functions from class GHS_ProjectManagerWindow.

FindFile()

```
FindFile(fileName, block=True, printOutput=True)
```

Locates an instance of the specified file or project in the Project Manager window and returns `True` on success and `False` on failure. This is analogous to searching for a file via the Project Manager's file shortcut bar when **Find** is enabled. For more information, see “Using the File Shortcut Bar” in Chapter 2, “The MULTI Project Manager” in the *MULTI: Building Applications* book.

Arguments are:

- `fileName` — Specifies the name of the file or project to locate.
- `block` — If `True`, executes `FindFile()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

The alias is: `Find()`

NextFile()

```
NextFile(fileName, block=True, printOutput=True)
```

Locates the next instance of the specified file or project in the Project Manager window and returns `True` on success and `False` on failure. This is analogous to searching for a file via the Project Manager's file shortcut bar when **Next** is enabled. For more information, see “Using the File Shortcut Bar” in Chapter 2, “The MULTI Project Manager” in the *MULTI: Building Applications* book.

Arguments are:

- `fileName` — Specifies the name of the file or project to locate.

- `block` — If `True`, executes `NextFile()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.
- `printOutput` — If `True`, prints the output (if any). If `False`, does not print output.

The alias is: `Next()`

GHS_ProjectManagerWindow Debug and Edit Functions

The following sections describe the functions from class `GHS_ProjectManagerWindow` that you can use to open projects in the MULTI Debugger or Editor.

DebugSelectedProjects()

`DebugSelectedProjects(block=False)`

Debugs selected projects, if applicable. This function returns `True` on success and `False` on failure.

The argument is:

- `block` — If `True`, executes `DebugSelectedProjects()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

Aliases are: `DebugSelected()`, `DebugSelectedProjs()`, `DebugSelectedProj()`, `DebugSel()`

EditSelectedProjects()

`EditSelectedProjects(block=True)`

Edits the selected project files. This function returns `True` on success and `False` on failure.

The argument is:

- `block` — If `True`, executes `EditSelectedProjects()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

Aliases are: `EditSelected()`, `EditSelectedProjs()`,
`EditSelectedProj()`, `EditSel()`

GHS_ProjectManagerWindow Tree Expansion/Contraction Functions

The following sections describe the GUI-related functions from class `GHS_ProjectManagerWindow`.

ContractAll()

`ContractAll(block=False)`

Contracts the entire project tree and returns `True` on success and `False` on failure.

The argument is:

- `block` — If `True`, executes `ContractAll()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

ContractSelected()

`ContractSelected(block=False)`

Contracts the selected project tree entries and returns `True` on success and `False` on failure.

The argument is:

- `block` — If `True`, executes `ContractSelected()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

The alias is: `ContractSel()`

ExpandAll()

`ExpandAll(block=False)`

Expands the entire project tree and returns `True` on success and `False` on failure.

The argument is:

- `block` — If `True`, executes `ExpandAll()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

ExpandSelected()

`ExpandSelected(block=False)`

Expands the selected project tree entries and returns `True` on success and `False` on failure.

The argument is:

- `block` — If `True`, executes `ExpandSelected()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

The alias is: `ExpandSel()`

PrintAll()

`PrintAll(block=True)`

Prints the fully expanded project tree (that is, expands the project tree and then prints it). This function returns `True` on success and `False` on failure.

The argument is:

- `block` — If `True`, executes `PrintAll()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

The alias is: `Print()` (Note that `print()` is not a valid alias for this function.)

PrintView()

`PrintView(block=True)`

Prints the parts of the project tree that have been expanded. This function returns `True` on success and `False` on failure.

The argument is:

- `block` — If `True`, executes `PrintView()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

GHS_ProjectManagerWindow Selection Functions

The following sections describe the functions from class `GHS_ProjectManagerWindow` that relate to project selection.

DoubleClickTreeRow()

`DoubleClickTreeRow(row, block=True)`

Simulates double-clicking the specified row in the project tree view. This function returns `True` on success and `False` on failure.

Arguments are:

- `row` — Specifies the index of the row to be double-clicked. The index starts at 0 (zero). If the row number is less than 0, all rows are selected.

- `block` — If `True`, executes `DoubleClickTreeRow()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.

The alias is: `DblClickTreeRow()`

SelectAll()

`SelectAll(block=True)`

Selects the entire project tree and returns `True` on success and `False` on failure.

The argument is:

- `block` — If `True`, executes `SelectAll()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

The alias is: `SelAll()`

SelectProject()

`SelectProject(projName, block=True)`

Selects an entry by its name. This function returns `True` on success and `False` on failure.

Arguments are:

- `projName` — Specifies the name of the entry to select.
- `block` — If `True`, executes `SelectProject()` in blocked mode and grabs and prints the output (if any). If `False`, neither executes the function in blocked mode nor grabs and prints the output.

The alias is: `SelProj()`

SelectTreeRow()

`SelectTreeRow(row, block=True)`

Selects a row in the project tree view and returns `True` on success and `False` on failure.

Arguments are:

- `row` — Specifies the index of the row to be selected. The index starts at 0 (zero). If the row number is less than 0, all rows are selected.
- `block` — If `True`, executes `SelectTreeRow()` in blocked mode and grabs the output (if any). If `False`, neither executes the function in blocked mode nor grabs the output.

The alias is: `SelTreeRow()`

Version Control Functions

Chapter 17

This Chapter Contains:

- GHS_CoBrowse Functions
- GHS_DiffView Functions
- GHS_DiffViewWindow Basic Functions
- GHS_DiffViewWindow Display Functions

This chapter documents functions from the following classes:

- `GHS_CoBrowse` (alias `ghs_cobrowse`) — Exposes the basic functions of the MULTI Checkout Browser service. This class inherits from class `GHS_IdeObject`. See “GHS_CoBrowse Functions” on page 330.
- `GHS_DiffView` (alias `ghs_diffview`) — Exposes the basic functions of the MULTI Diff Viewer service. This class inherits from class `GHS_IdeObject`. See “GHS_DiffView Functions” on page 331.
- `GHS_DiffViewWindow` — Exposes additional functions of the MULTI Diff Viewer. This class inherits from class `GHS_Window`.

The `GHS_DiffViewWindow` functions are divided into the following sections:

- “GHS_DiffViewWindow Basic Functions” on page 333
- “GHS_DiffViewWindow Display Functions” on page 334

GHS_CoBrowse Functions

The following sections describe functions from class `GHS_CoBrowse`.

`__init__()`

`__init__(workingDir="")`

Initializes the object attributes.

The argument is:

- `workingDir` — Stores the working directory of the MULTI service object.

OpenCheckoutBrowserWindow()

```
OpenCheckoutBrowserWindow(checkoutDir="",  
scanNow=False)
```

Opens a MULTI Checkout Browser window on the given directory. On success, this function returns a GHS_CoBrowseWindow object for the MULTI Checkout Browser window. On failure, it returns None.

Arguments are:

- checkoutDir — Specifies the directory to examine.
- scanNow — If True, immediately scans the file checkout information. If False, does not immediately scan the file checkout information.

Aliases are: OpenCheckoutBrowserWin(),
OpenCoBrowseWindow(), OpenCoBrowseWin(),
OpenCoBrowse(), OpenWindow(), OpenWin()

GHS_DiffView Functions

The following sections describe functions from class GHS_DiffView.

__init__()

```
__init__(workingDir="")
```

Initializes the object attributes.

The argument is:

- workingDir — Stores the working directory of the MULTI service object.

DiffFiles()

```
DiffFiles(fileName1, fileName2, reuse=False,  
append=False)
```

Compares two files and displays their differences in a MULTI Diff Viewer window. This function returns a `GHS_DiffViewWindow` object for the newly created MULTI Diff Viewer window, or it returns `None`.

Arguments are:

- `fileName1` — Specifies the name of the first file.
- `fileName2` — Specifies the name of the second file.
- `reuse` — If `True`, reuses an existing MULTI Diff Viewer window. If `False`, opens a new window.
- `append` — indicates if only to append an entry for the compared files into an existing MULTI Diff Viewer window without switching to show the differences of the compared files. This argument is effective only if `reuse` is `True`.

OpenChooseWindow()

```
OpenChooseWindow(fileName1="", fileName2="",  
reuse=True)
```

Opens a dialog box in which you can specify two files for comparison. This function returns a `GHS_Window` object for the dialog box, or it returns `None` on failure.

Arguments are:

- `fileName1` — Specifies the first filename. If specified, the filename appears in the dialog box.
- `fileName2` — Specifies the second filename. If specified, the filename appears in the dialog box.
- `reuse` — If `True`, reuses an existing MULTI Diff Viewer window when you ask to compare the specified files. If `False`, opens a new Diff Viewer window.

The alias is: `OpenChooseWin()`

GHS_DiffViewWindow Basic Functions

The following sections describe the basic functions from class `GHS_DiffViewWindow`.

OpenDiff()

`OpenDiff()`

Opens a window so that you can specify files for comparison. When you click the **Diff** button in the window, the current MULTI Diff Viewer is reused to compare the specified files.

Returns a `GHS_Window` object for the window for you to specify files to compare and `None` on failure.

OpenNewDiff()

`OpenNewDiff()`

Opens a dialog box in which you can specify two files for comparison. When you click the dialog box's **Diff** button, a new MULTI Diff Viewer window is created to compare the specified files. This function returns a `GHS_Window` object for the dialog box, or it returns `None` on failure.

ShowCurrentDiff()

`ShowCurrentDiff()`

Repositions the MULTI Diff Viewer pane to display the current difference (this is useful if you lose your place while scrolling). This function returns `True` on success and `False` on failure.

The alias is: `ShowCurDiff()`

GHS_DiffViewWindow Display Functions

The following sections describe the display functions from class `GHS_DiffViewWindow`.

ToggleCaseSensitive()

`ToggleCaseSensitive()`

Toggles the flag for ignoring case differences between two lines. This function returns `True` on success and `False` on failure.

For more information about the flag, see **Case Insensitive** in “Diff Viewer Menus” in Chapter 7, “Version Control Tools” in the *MULTI: Editing Files and Configuring the IDE* book.

The alias is: `Case()`

ToggleCharDiff()

`ToggleCharDiff()`

Toggles the flag for displaying character changes. This function returns `True` on success and `False` on failure.

For more information about the flag, see **Display Changes Within Lines** in “Diff Viewer Menus” in Chapter 7, “Version Control Tools” in the *MULTI: Editing Files and Configuring the IDE* book.

The alias is: `CharDiff()`

ToggleIgnoreAllWhiteSpace()

`ToggleIgnoreAllWhiteSpace()`

Toggles the flag for ignoring all whitespace differences between two files. This function returns `True` on success and `False` on failure.

For more information about the flag, see **Ignore All Whitespace** in “Diff Viewer Menus” in Chapter 7, “Version Control Tools” in the *MULTI: Editing Files and Configuring the IDE* book.

The alias is: `IgnoreAllSpace`

ToggleIgnoreCWhiteSpace()

`ToggleIgnoreCWhiteSpace()`

Toggles the flag for ignoring C whitespace differences between two files. This function returns `True` on success and `False` on failure.

For more information about the flag, see **Ignore Whitespace for C** in “Diff Viewer Menus” in Chapter 7, “Version Control Tools” in the *MULTI: Editing Files and Configuring the IDE* book.

The alias is: `IgnoreCSpace()`

ToggleIgnoreWhiteSpaceAmount()

`ToggleIgnoreWhiteSpaceAmount()`

Toggles the flag for ignoring differences in the amounts of whitespace between two lines. This function returns `True` on success and `False` on failure.

For more information about the flag, see **Ignore Whitespace Amount** in “Diff Viewer Menus” in Chapter 7, “Version Control Tools” in the *MULTI: Editing Files and Configuring the IDE* book.

The alias is: `IgnoreSpaceAmount()`

ToggleLineupColumns()

`ToggleLineupColumns()`

Toggles the flag for lining up columns to determine differences within lines. This function returns `True` on success and `False` on failure.

For more information about the flag, see **Line up Columns for Changes Within Lines** in “Diff Viewer Menus” in Chapter 7, “Version Control Tools” in the *MULTI: Editing Files and Configuring the IDE* book.

The alias is: `LineupCol()`

ToggleNumber()

`ToggleNumber()`

Toggles the flag for ignoring number differences between lines. This function returns `True` on success and `False` on failure.

For more information about the flag, see **Ignore Changes in Numbers (0–9)** in “Diff Viewer Menus” in Chapter 7, “Version Control Tools” in the *MULTI: Editing Files and Configuring the IDE* book.

Aliases are: `ToggleNum()`, `NumDiff()`

ToggleWordDiff()

`ToggleWordDiff()`

Toggles the flag for displaying line differences word by word. This function returns `True` on success and `False` on failure.

For more information about the flag, see **Display Changes Word by Word Within Lines** in “Diff Viewer Menus” in Chapter 7, “Version Control Tools” in the *MULTI: Editing Files and Configuring the IDE* book.

The alias is: `WordDiff()`

Miscellaneous Functions

Chapter 18

This Chapter Contains:

- GHS_AbortExecFile and GHS_AbortExecFileWithStack Functions
- GHS_AbortExecOnSignal Functions
- GHS_Exception Functions
- GHS_WindowClassNames Attributes and Functions

This chapter documents functions from the following classes:

- `GHS_AbortExecFile` — Aborts file script execution via the utility function `GHS_ExecFile()` (see “`GHS_ExecFile()`” on page 92). This class inherits from class `GHS_Exception`. See “`GHS_AbortExecFile` and `GHS_AbortExecFileWithStack` Functions” on page 339.
- `GHS_AbortExecFileWithStack` — Aborts file script execution and prints the stack via the utility function `GHS_ExecFile()` (see “`GHS_ExecFile()`” on page 92). This class inherits from class `GHS_Exception`. See “`GHS_AbortExecFile` and `GHS_AbortExecFileWithStack` Functions” on page 339.
- `GHS_AbortExecOnSignal` — Aborts Python execution on signal. This class inherits from class `GHS_Exception`. See “`GHS_AbortExecOnSignal` Functions” on page 339.
- `GHS_Exception` — Describes MULTI-Python exceptions. This class inherits from the built-in Python class `Exception`. See “`GHS_Exception` Functions” on page 340.
- `GHS_WindowClassNames` — Defines the window class names supported in the MULTI IDE. See “`GHS_WindowClassNames` Attributes and Functions” on page 340.



Note The `GHS_Exception`, `GHS_AbortExecFile`, `GHS_AbortExecFileWithStack`, and `GHS_AbortExecOnSignal` classes are all used to stop script execution. If the script is executed by `GHS_ExecFile`, it is handled more gracefully than if it is executed by the built-in Python function `execfile()`. If the script is executed by `execfile()`, the execution stack is dumped when execution stops.

GHS_AbortExecFile and GHS_AbortExecFileWithStack Functions

The following section describes the `__init__()` function from classes `GHS_AbortExecFile` and `GHS_AbortExecFileWithStack`.

`__init__()`

`__init__(value=None)`

Initializes the object attributes.

The argument is:

- `value` — Stores any value to be printed out when script execution stops. The value is commonly a string.

GHS_AbortExecOnSignal Functions

The following section describes the `__init__()` function from class `GHS_AbortExecOnSignal`.

`__init__()`

`__init__(value=None)`

Initializes the object attributes.

The argument is:

- `value` — Usually stores a signal name or number (but can store any value) to be printed out when script execution stops. If you store an integer, it is interpreted as a signal number and the signal name is printed out. Python trace information is always printed out.

GHS_Exception Functions

The following section describes the `__init__()` function from class `GHS_Exception`.

`__init__()`

`__init__(value=None)`

Initializes the object attributes.

The argument is:

- `value` — Stores any value to be printed out when script execution stops. The value is commonly a string.

GHS_WindowClassNames Attributes and Functions

The class `GHS_WindowClassNames` defines window class names supported in the MULTI IDE.

The following list describes the attributes of this class:

- `coBrowser` — Stores the Checkout Browser's window class name.
- `connectionOrganizer` — Stores the Connection Organizer's window class name. The Connection Organizer is used in the Debugger.
- `debugger` — Stores the Debugger's window class name.
- `dialog` — Stores the dialog's window class name.
- `diffView` — Stores the Diff Viewer's window class name.
- `editor` — Stores the Editor's window class name.
- `eventAnalyzer` — Stores the EventAnalyzer's window class name.
- `helpViewer` — Stores the Help Viewer's window class name.
- `launcher` — Stores the Launcher's window class name.
- `misc` — Stores window class names for miscellaneous windows.
- `osaExplorer` — Stores the OSA Explorer's window class name. The OSA Explorer is used in the Debugger.

- `projectManager` — Stores the Project Manager's window class name.
- `pythonGui` — Stores the window class name for the stand-alone **Python GUI**.
- `resourceAnalyzer` — Stores the ResourceAnalyzer's window class name.
- `taskManager` — Stores the Task Manager's window class name. The Task Manager is used in the Debugger.
- `terminal` — Stores the Serial Terminal's window class name.
- `trace` — Stores the Trace List's window class name. The Trace List is used in the Debugger.

The following section describes the function from class `GHS_WindowClassNames`.

`__init__()`

`__init__()`

Initializes the object.

Appendix

Part IV

Third-Party License and Copyright Information

Appendix A

This Appendix Contains:

- PSF License Agreement for Python 2.3
- Tcl/Tk License Terms
- BLT Copyright Information

This appendix contains licensing and copyright information for third-party tools shipped with the MULTI IDE.



Note The MULTI installation does not use the standard Python interpreter. Instead, it uses a Green Hills interpreter that extends the standard Python interpreter and supports communication with the Green Hills toolchain.



Note A few Python script files in the Python installation that is shipped with MULTI have been modified to fix bugs.

PSF License Agreement for Python 2.3

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.3 software in source or binary form and its associated documentation.
2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.3 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright (c) 2001, 2002, 2003 Python Software Foundation; All Rights Reserved" are retained in Python 2.3 alone or in any derivative version prepared by Licensee.
3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.3 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.3.
4. PSF is making Python 2.3 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.3 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.3 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.3, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.
6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By copying, installing or otherwise using Python 2.3, Licensee agrees to be bound by the terms and conditions of this License Agreement.

Tcl/Tk License Terms

This software is copyrighted by the Regents of the University of California, Sun Microsystems, Inc., Scriptics Corporation, and other parties. The following terms apply to all files associated with the software unless explicitly disclaimed in individual files.

The authors hereby grant permission to use, copy, modify, distribute, and license this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. No written agreement, license, or royalty fee is required for any of the authorized uses. Modifications to this software may be copyrighted by their authors and need not follow the licensing terms described here, provided that the new terms are clearly indicated on the first page of each file where they apply.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NON-INFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN “AS IS” BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

GOVERNMENT USE: If you are acquiring this software on behalf of the U.S. government, the Government shall have only “Restricted Rights” in the software and related documentation as defined in the Federal Acquisition Regulations (FARs) in Clause 52.227.19 (c) (2). If you are acquiring the software on behalf of the Department of Defense, the software shall be classified as “Commercial Computer Software” and the Government shall have only “Restricted Rights” as defined in Clause 252.227-7013 (c) (1) of DFARs. Notwithstanding the foregoing, the authors grant the U.S. Government and others acting in its behalf permission to use and distribute the software in accordance with the terms specified in this license.

BLT Copyright Information



Note Each filename listed in this section is followed by its copyright and license, if any.

bltCanvEps.pro

Copyright 1991-1997 Bell Labs Innovations for Lucent Technologies.

dnd.tcl

Copyright (c) 1998 Lucent Technologies, Inc.

dragdrop.tcl

Copyright (c) 1998 Lucent Technologies, Inc.

hierbox.tcl

Copyright (c) 1998 Lucent Technologies, Inc.

tabnotebook.tcl

Copyright (c) 1998 Lucent Technologies, Inc.

tabset.tcl

Copyright (c) 1998 Lucent Technologies, Inc.

treeview.tcl

Copyright (c) 1998 Lucent Technologies, Inc.

All of the preceding files contain the following license:

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that the copyright notice and warranty disclaimer appear in supporting documentation, and that the names of Lucent Technologies any of their entities not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

Lucent Technologies disclaims all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall Lucent Technologies be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortuous action, arising out of or in connection with the use or performance of this software.

bltGraph.pro

Copyright 1989-1992 Regents of the University of California.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies. The University of California makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Copyright 1991-1997 Bell Labs Innovations for Lucent Technologies.

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that

the above copyright notice appear in all copies and that both that the copyright notice and warranty disclaimer appear in supporting documentation, and that the names of Lucent Technologies any of their entities not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

Lucent Technologies disclaims all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall Lucent Technologies be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortuous action, arising out of or in connection with the use or performance of this software.

dd-color.tcl

Copyright (c) 1993 AT&T All Rights Reserved

dd-file.tcl

Copyright (c) 1993 AT&T All Rights Reserved

dd-number.tcl

Copyright (c) 1993 AT&T All Rights Reserved

dd-text.tcl

Copyright (c) 1993 AT&T All Rights Reserved

stripchart1.tcl

Copyright (c) 1996 Lucent Technologies

Index

A

- AddAction(), 301
- additional Python modules
 - installing, 31
- AddStr(), 268
- AddString(), 268
- AddVar(), 308
- AddVariable(), 307
- AddWorkspace(), 303
- AddWs(), 303
- Alive(), 98
- args** command line option, 53
- Attach(), 260
- attributes
 - GHS_Action, 291
 - GHS_ActionSequence, 292
 - GHS_DebuggerApi, 238
 - GHS_IdeObject, 32
 - GHS_MemorySpaces, 255
 - GHS_MslTree, 141
 - GHS_OsTypes, 256
 - GHS_Variable, 293
 - GHS_Window, 34
 - GHS_WindowClassNames, 340
 - GHS_Workspace, 294
- AutoTimeUnit(), 286
- AutoUnit(), 286

B

- basic functions, 98
- Beep()
 - GHS_Window class, 108

- GHS_WindowRegister class, 201
- BigEndianTarget(), 245
- blocked mode, 90
- board setup scripts, 16, 24
- bpsyntaxchecking** configuration
 - option, 23
- Build(), 315
- BuildAll(), 314
- BuildAllProj(), 314
- BuildAllProjects(), 314
- BuildAllProjs(), 314
- BuildFile(), 314
- BuildProj(), 315
- BuildProjects(), 315
- BuildProjs(), 315
- BuildSel(), 316
- BuildSelected(), 316
- BuildSelectedProj(), 316
- BuildSelectedProjects(), 316
- BuildSelectedProjs(), 316

C

- \$c** Py pane command, 50
- Case(), 334
- ChangeB(), 227
- ChangeBaudRate(), 220
- ChangeBpInheritance(), 227
- ChangeBreakpointInheritance(), 226
- ChangeC(), 227
- ChangeD(), 228
- ChangeDebugChildren(), 227
- ChangeDebugOnTaskCreation(), 227
- ChangeE(), 230

Index

- ChangeF(), 231
- ChangeH(), 228
- ChangeHaltOnAttach(), 228
- ChangeI(), 229
- ChangeInheritProcessBits(), 229
- ChangeMslTree(), 156
- ChangePdVal(), 170
- ChangePdValue(), 170
- ChangePullDownValue(), 170
- ChangeR(), 230
- ChangeRunOnDetach(), 229
- ChangeStopAfterExec(), 230
- ChangeStopAfterFork(), 230
- ChangeStopOnTaskCreation(), 231
- ChangeT(), 231
- ChangeTextFieldValue(), 183
- ChangeTfVal(), 183
- ChangeTfValue(), 183
- ChangeTimeUnit(), 286
- ChangeUnit(), 286
- ChangeVar(), 308
- ChangeVariable(), 308
- ChangeWholeMslTree(), 157
- CharDiff(), 334
- CheckB(), 232
- CheckBpInheritance(), 232
- CheckBreakpointInheritance(), 232
- CheckC(), 232
- CheckD(), 232
- CheckDebugChildren(), 232
- CheckDebugOnTaskCreation(), 232
- CheckE(), 234
- CheckF(), 234
- CheckH(), 233
- CheckHaltOnAttach(), 233
- CheckI(), 233
- Checkin(), 276
- CheckInheritProcessBits(), 233
- Checkout(), 276
- CheckR(), 233
- CheckRunOnDetach(), 233
- CheckStopAfterExec(), 234
- CheckStopAfterFork(), 234
- CheckStopOnTaskCreation(), 234
- CheckSym(), 244
- CheckSymbol(), 244
- CheckT(), 234
- CheckWin(), 191
- CheckWindow(), 191
- CheckWindowObj(), 192
- CheckWindowObject(), 192
- CheckWinObj(), 192
- ChooseDir()
 - GHS_Window class, 108
 - GHS_WindowRegister class, 201
- ChooseFile()
 - GHS_Window class, 109
 - GHS_WindowRegister class, 202
- ChooseFromList()
 - GHS_Window class, 110
 - GHS_WindowRegister class, 203
- ChooseMenu(), 120
- ChooseSubMenu(), 121
- ChooseSubSubMenu(), 122
- ChooseWin()
 - GHS_Window class, 111
 - GHS_WindowRegister class, 204
- ChooseWindow()
 - GHS_Window class, 111
 - GHS_WindowRegister class, 204
- ChooseWindowFromGui()
 - GHS_Window class, 110
 - GHS_WindowRegister class, 203
- ChooseYesNo()
 - GHS_Window class, 111
 - GHS_WindowRegister class, 204
- classes, MULTI-Python, 32
 - Debugger object classes, 36
 - GHS_AbortExecFile, 338
 - GHS_AbortExecFileWithStack, 338
 - GHS_AbortExecOnSignal, 338
 - GHS_Action, 290

Index

- GHS_ActionSequence, 290
- GHS_CoBrowse, 330
- GHS_ConnectionOrganizerWindow, 278
- GHS_Debugger, 224
- GHS_DebuggerApi, 212, 224
- GHS_DebuggerWindow, 224
- GHS_DebugServer, 212
- GHS_DiffView, 330
- GHS_DiffViewWindow, 330
- GHS_Editor, 266
- GHS_EditorWindow, 266
- GHS_EventAnalyzerWindow, 278
- GHS_Exception, 338
- GHS_IdeObject, 32, 98
- GHS_Launcher, 290
- GHS_LauncherWindow, 290
- GHS_MemorySpaces, 224
- GHS_MslTree, 140
- GHS_OsTypes, 224
- GHS_ProjectManager, 312
- GHS_ProjectManagerWindow, 312
- GHS_TargetIds, 225
- GHS_Task, 225
- GHS_Terminal, 212
- GHS_TerminalWindow, 212
- GHS_TraceWindow, 225
- GHS_Variable, 290
- GHS_Window, 34, 100, 140
- GHS_WindowClassNames, 338
- GHS_WindowRegister, 190
- GHS_Workspace, 290
- hierarchy of, 37
- miscellaneous classes, 37
- overview, 32
- service classes, 33
- utility classes, 36
- window classes, 35
- CleanCmdExecVariables(), 98
- \$clear** Py pane command, 50
- ClearConfigFile(), 103
- ClearDefaultConfigFile(), 103
- ClearDftConfigFile(), 103
- CloseAllWindows(), 206
- CloseCurFile(), 271
- CloseCurrentFile(), 271
- CloseFile()
 - GHS_EditorWindow class, 271
 - GHS_EventAnalyzer class, 279
 - GHS_EventAnalyzerWindow class, 281
- CloseProj(), 319
- CloseProject(), 318
- CloseWin(), 133
- CloseWindow(), 133
- CloseWindows(), 206
- CloseWins(), 206
- CmdToChangeMslTree(), 161
- CmdToChangePdVal(), 171
- CmdToChangePdValue(), 171
- CmdToChangeTfVal(), 184
- CmdToChangeTfValue(), 184
- CmdToClickBut(), 152
- CmdToClickButton(), 152
- CmdToDbClickMslCell(), 162
- CmdToDbClickMslCellByVal(), 162
- CmdToDbClickMslCellByValue(), 162
- CmdToDumpAll(), 148
- CmdToDumpBut(), 152
- CmdToDumpButton(), 152
- CmdToDumpMenu(), 115
- CmdToDumpMenuBar(), 115
- CmdToDumpMsl(), 163
- CmdToDumpMslHl(), 163
- CmdToDumpMslSel(), 164
- CmdToDumpPdMenu(), 172
- CmdToDumpPdVal(), 172
- CmdToDumpPdValue(), 172
- CmdToDumpTab(), 177
- CmdToDumpTabSel(), 177
- CmdToDumpTabVal(), 178
- CmdToDumpTf(), 185
- CmdToDumpTx(), 180

Index

- CmdToDumpWidget(), 148
 - CmdToDumpWin(), 148
 - CmdToExtMslSel(), 165
 - CmdToRegDlgCmd(), 124
 - CmdToRegDlgCmds(), 124
 - CmdToRetOnTf(), 185
 - CmdToReturnOnTf(), 185
 - CmdToSelMenu(), 116
 - CmdToSelMenuPath(), 116
 - CmdToSelMslCell(), 165
 - CmdToSelMslCellByVal(), 166
 - CmdToSelMslCellByValue(), 166
 - CmdToSelPdMenu(), 173
 - CmdToSelSubMenu(), 117
 - CmdToSelSubSubMenu(), 118
 - CmdToSelTab(), 178
 - CmdToShowWidgets(), 149
 - CmdToSortMsl(), 167
 - collecting execution information
 - example MULTI script, 20
 - command help, 6
 - command line options
 - mpythonrun, 53
 - commands
 - help regarding, 6
 - helpview**, 6
 - Py pane, 50
 - sc**, 23
 - socket server, 57
 - compatibility
 - MULTI-Python, 32
 - configuration options
 - bpsyntaxchecking**, 23
 - help regarding, 6
 - Connect()
 - GHS_DebuggerApi class, 215
 - GHS_Terminal class, 219
 - GHS_TerminalWindow class, 220
 - Connected(), 215
 - connecting to a target
 - example MULTI script, 18
 - connection functions, 212
 - ConnectRtserve(), 213
 - ConnectRtserve2(), 214
 - ConnectTarget(), 215
 - ConnectToRtserve(), 213
 - ConnectToRtserve2(), 214
 - ConnectToTarget(), 214
 - context-sensitive help, 6
 - ContractAll(), 323
 - ContractSel(), 324
 - ContractSelected(), 323
 - conventions
 - typographical, xxii
 - CoProcessor(), 246
 - Copy()
 - GHS_EditorWindow class, 268
 - CopySel(), 317
 - CopySelected(), 317
 - CpuFamily(), 246
 - CpuMinor(), 246
 - CpuName(), 247
 - cr** command line option, 53
 - CreateWorkspace(), 303
 - CreateWs(), 303
 - creating graphical interfaces, 58, 85
 - Ctrl+Enter Py pane keyboard shortcut, 52
 - Ctrl+h Py pane keyboard shortcut, 52
 - Ctrl+i Py pane keyboard shortcut, 52
 - Ctrl+s Py pane keyboard shortcut, 52
 - Ctrl+Shift+i Py pane keyboard shortcut, 52
 - Ctrl+Shift+s Py pane keyboard shortcut, 52
 - customized Python installation, 31
 - Cut(), 269
 - CutSel(), 318
 - CutSelected(), 318
 - Cwd(), 101
- ## D
- \$d** Py pane command, 50

Index

DblClickMslCell(), 158
DblClickMslCellByVal(), 158
DblClickMslCellByValue(), 158
DblClickTreeRow(), 326
debug functions, 224
debug servers
 commands, 17
Debug(), 239
DebugFile(), 239
DebuggedProgram(), 240
Debugger object classes, 36
DebugProg(), 239
DebugProgram(), 239
DebugSel(), 322
DebugSelected(), 322
DebugSelectedProj(), 322
DebugSelectedProjects(), 322
DebugSelectedProjs(), 322
default Python installation, 30
DelAction(), 302
DeleteAction(), 302
DeleteSelected(), 318
DeleteVariable(), 308
DeleteWorkspace(), 303
DelSel(), 318
DelSelected(), 318
DelVar(), 309
DelWs(), 304
Detach(), 260
DiffFiles(), 332
DirChooser()
 GHS_Window class, 108
 GHS_WindowRegister class, 202
Disconnect()
 GHS_DebuggerApi class, 215
 GHS_DebugServer class, 217
 GHS_TerminalWindow class, 220
\$display Py pane command, 50
DisplayMessage()
 GHS_Window class, 112
 GHS_WindowRegister class, 206
DisplayMsg()
 GHS_Window class, 112
 GHS_WindowRegister class, 206
document set, xx to xxi
DoubleClickMslCell(), 157
DoubleClickMslCellByValue(), 158
DoubleClickTreeRow(), 325
Dump()
 GHS_Action class, 292
 GHS_ActionSequence class, 292
 GHS_MslTree class, 142
 GHS_Variable class, 294
 GHS_Workspace class, 295
DumpAll(), 146
DumpBut(), 151
DumpButton(), 151
DumpMenu(), 113
DumpMenuBar(), 114
DumpMsl(), 160
DumpMslHighlight(), 158
DumpMslHl(), 159
DumpMslSel(), 159
DumpMslSelection(), 159
DumpMslValue(), 159
DumpPdMenu(), 170
DumpPdValue(), 171
DumpPullDownMenu(), 170
DumpPullDownValue(), 171
DumpTab(), 175
DumpTabContents(), 175
DumpTabSel(), 176
DumpTabSelection(), 175
DumpTabVal(), 176
DumpTabValue(), 176
DumpTcVal(), 182
DumpTcValue(), 182
DumpTextCellValue(), 181
DumpTextFieldValue(), 183
DumpTfVal(), 184
DumpTfValue(), 184
DumpToFile(), 254

Index

DumpTree(), 142
 class GHS_Action, 291
 class GHS_ActionSequence, 292
 class GHS_Variable, 293
 class GHS_Workspace, 295
DumpWidget(), 147
DumpWin(), 147
DumpWindow(), 147

E

\$e Py pane command, 50
EditFile()
 GHS_Editor class, 267
 GHS_EditorWindow class, 272
EditLines(), 155
Editor functions, 266
EditSel(), 323
EditSelected(), 323
EditSelectedProj(), 323
EditSelectedProjects(), 322
EditSelectedProjs(), 323
EditTextLines(), 155
EditTextStr(), 156
EditTextString(), 156
environment, MULTI-Python
 extending, 46
Esc Py pane keyboard shortcut, 52
EventAnalyzer functions, 278
examples
 MULTI scripts, 18
 MULTI-Python
 manipulating Debugger, 77
 manipulating Editor, 75
 manipulating windows, 62
ExecCmd(), 226
ExecCmds(), 226
\$execute Py pane command, 50
ExecuteCmd(), 226
ExecuteCmds(), 226
Expandable(), 143

ExpandAll(), 324
Expanded(), 143
ExpandSel(), 324
ExpandSelected(), 324
extending MULTI-Python environment, 46
ExtendMslSelection(), 160
ExtMslSel(), 160

F

-f command line option, 54
F1 key, for help, 6
FileChooser()
 GHS_Window class, 109
 GHS_WindowRegister class, 202
Files(), 279
Find(), 321
FindFile(), 321
FirstView(), 282
Flash(), 273
FlashCursor(), 273
FlatView(), 284
flow control statements
 in MULTI scripts, 16
FlushTraceBuf(), 263
FlushTraceBuffer(), 263
functions, MULTI-Python
 basic, 98
 connection, 212
 debug, 224
 Editor, 266
 EventAnalyzer, 278
 general notes on using, 90
 GHS_AbortExecFile, 339
 GHS_AbortExecFileWithStack, 339
 GHS_AbortExecOnSignal, 339
 GHS_Action, 291
 GHS_ActionSequence, 292
 GHS_CoBrowse, 330
 GHS_Debugger, 225
 GHS_DebuggerApi

Index

- debug flag, 226
- host information, 235
- memory access, 236
- run control, 239
- symbol, 244
- target connection, 213
- target information, 245
- window display, 216, 249
- GHS_DebuggerWindow
 - basic, 250
 - breakpoint, 251
 - print, 254
- GHS_DebugServer, 216
- GHS_DiffView, 331
- GHS_DiffViewWindow
 - basic, 333
 - display, 334
- GHS_Editor, 266
- GHS_EditorWindow
 - edit, 268
 - file, 271
 - selection and cursor, 273
 - version control, 276
- GHS_EventAnalyzer, 278
- GHS_EventAnalyzerWindow
 - file, 281
 - miscellaneous, 286
 - view and selection, 282
- GHS_Exception, 340
- GHS_IdeObject, 98
- GHS_Launcher, 297
- GHS_LauncherWindow
 - action execution, 298
 - action manipulation, 301
 - variable, 307
 - workspace manipulation, 303
- GHS_MemorySpaces, 255
- GHS_MslTree, 142
- GHS_OsTypes, 256
- GHS_ProjectManager, 312
- GHS_ProjectManagerWindow
 - build, 314
 - debug and edit, 322
 - edit, 317
 - file, 318
 - navigation, 321
 - selection, 325
 - tree expansion/contraction, 323
- GHS_TargetIds, 257
- GHS_Task
 - basic, 258
 - run control, 260
- GHS_Terminal, 219
- GHS_TerminalWindow, 220
- GHS_TraceWindow, 263
- GHS_Variable, 293
- GHS_Window
 - basic, 101
 - basic widget, 146
 - Button widget, 151
 - configuration, 103
 - directory, 105
 - Edit and Terminal widget, 155
 - interactive, 108
 - menu, 113
 - modal dialog, 123
 - MScrollList widget, 156
 - PullDown widget, 170
 - record, 132
 - Tab widget, 174
 - Text widget, 180
 - TextCell widget, 181
 - TextField widget, 183
 - window attribute and manipulation, 133
- GHS_WindowClassNames, 341
- GHS_WindowRegister
 - basic, 190
 - check, 191
 - get window, 193
 - interactive, 201
 - wait, 207

Index

- window manipulation, 206
- GHS_Workspace, 295
- Launcher, 290
- miscellaneous, 338
- Project Manager, 312
- version control, 330
- widget, 140
- window, 100
- window tracking, 190

G

general notes on using functions, 90

- GetBuilder(), 197
- GetBuilderWin(), 197
- GetBuilderWindow(), 197
- GetCheckoutBrowser(), 193
- GetCheckoutBrowserWindow(), 193
- GetChildNum(), 142
- GetChildrenNumber(), 142
- GetCo(), 194
- GetCoB(), 193
- GetCommandToChangeMslTree(), 161
- GetCommandToChangePullDownValue(), 171
- GetCommandToChangeTextFieldValue(), 184
- GetCommandToClickButton(), 152
- GetCommandToDoubleClickMslCell(), 161
- GetCommandToDoubleClickMslCell...(), 162
- GetCommandToDumpButton(), 152
- GetCommandToDumpMenu(), 114
- GetCommandToDumpMenuBar(), 115
- GetCommandToDumpMsl(), 163
- GetCommandToDumpMslHighlight(), 163
- GetCommandToDumpMslSelection(), 164
- GetCommandToDumpPullDownMenu(), 172
- GetCommandToDumpPullDownValue(), 172
- GetCommandToDumpTab(), 176
- GetCommandToDumpTabSelection(), 177
- GetCommandToDumpTabValue(), 177
- GetCommandToDumpText(), 180
- GetCommandToDumpTextField(), 185
- GetCommandToDumpWidget(), 147

- GetCommandToDumpWindow(), 148
- GetCommandToExtendMslSelection(), 164
- GetCommandToRegisterModalDialog...(), 123
- GetCommandToReturnOnTextField(), 185
- GetCommandToSelectMenu(), 115
- GetCommandToSelectMenuPath(), 116
- GetCommandToSelectMslCell(), 165
- GetCommandToSelectMslCellByValue(), 166
- GetCommandToSelectPullDownMenu(), 173
- GetCommandToSelectSubMenu(), 116
- GetCommandToSelectSubSubMenu(), 117
- GetCommandToSelectTab(), 178
- GetCommandToShowWidgets(), 149
- GetCommandToSortMsl(), 166
- GetComponent(), 217
- GetConnectionOrganizer(), 194
- GetConnectionOrganizerWindow(), 194
- GetCpuFamily(), 246
- GetCpuFamilyName(), 247
- GetCpuMinor(), 246
- GetCpuName(), 247
- GetCurrentTab(), 179
- GetCurTab(), 179
- GetCwd(), 101
- GetDebugger(), 194
- GetDebuggerWin(), 194
- GetDebuggerWindow(), 194
- GetDialog(), 194
- GetDialogByName(), 194
- GetDiffViewer(), 195
- GetDiffViewerWindow(), 195
- GetDim(), 133
- GetDimension(), 133
- GetDlg(), 194
- GetDlgByName(), 194
- GetDv(), 195
- GetEditor(), 195
- GetEditorWin(), 195
- GetEditorWindow(), 195
- GetEditTextLines(), 155
- GetEditTextString(), 155

Index

GetEventAnalyzerWin(), 195
GetEventAnalyzerWindow(), 195
GetFileList(), 279
GetFiles(), 279
GetHelpViewer(), 196
GetHelpViewerWindow(), 196
GetHostOsName(), 235
GetHv(), 196
GetInfo(), 101
GetInput()
 GHS_Window class, 111
 GHS_WindowRegister class, 205
GetIntDir(), 105
GetIntDistDir(), 105
GetIntegrityDistributionDir(), 105
\$getinteractive Py pane command, 50
GetLatestDir(), 106
GetLauncher(), 196
GetLauncherWin(), 196
GetLauncherWindow(), 196
GetMea(), 195
GetMeaWin(), 195
GetMeaWindow(), 195
GetMev(), 195
GetMra(), 197
GetMruDir(), 107
GetMrv(), 197
GetMslRowNum(), 167
GetMslRowNumber(), 167
GetMslTree(), 167
GetMultiVersion(), 235
GetName(), 133
GetOsa(), 196
GetOsaExplorer(), 196
GetOsaExplorerWindow(), 196
GetOsName(), 248
GetPc(), 240
GetPdMenu(), 173
GetPdVal(), 174
GetPdValue(), 174
GetPid(), 101
GetPos(), 134
GetPosition(), 134
GetProgram(), 240
GetProjectManagerWindow(), 197
GetProjMgrWin(), 197
GetPullDownMenu(), 173
GetPullDownValue(), 174
GetPyGui(), 197
GetPythonGuiWindow(), 197
GetResourceAnalyzer(), 197
GetResourceAnalyzerWindow(), 197
GetRunningActions(), 298
GetSel(), 274
GetSelectedString(), 274
GetSelection(), 273
GetSelStr(), 274
GetSeries(), 248
GetStatus()
 GHS_DebuggerApi class, 240
 GHS_ProjectManagerWindow
 class, 316
GetSymAdr(), 245
GetSymbolAddress(), 245
GetSymbolSize(), 245
GetSymSize(), 245
GetTabNames(), 179
GetTabSel(), 179
GetTabSelection(), 179
GetTargetCoProcessor(), 246
GetTargetCpuFamilyName(), 247
GetTargetId(), 247
GetTargetOs(), 248
GetTargetOsMinor(), 247
GetTargetOsMinorType(), 247
GetTargetOsName(), 247
GetTargetOsType(), 248
GetTargetPid(), 240
GetTargetSeries(), 248
GetTaskManagerWin(), 198
GetTaskManagerWindow(), 198
GetTcVal(), 182

Index

GetTcValue(), 182
GetTerm(), 198
GetTerminal(), 198
GetTerminalWindow(), 198
GetTermTextLines(), 155
GetTermTextString(), 156
GetTextCellValue(), 182
GetTextFieldValue(), 186
GetTextLines(), 269
GetTextString(), 269
GetTextValue(), 181
GetTfVal(), 186
GetTfValue(), 186
GetTm(), 198
GetTopProjectFiles(), 313
GetTrace(), 198
GetTraceWindow(), 198
GetTxVal(), 181
GetUvelDir(), 106
GetUvelDistDir(), 106
GetUvelosityDistributionDir(), 106
GetWin(), 199
GetWinByIdx(), 199
GetWinByName(), 199
GetWindow(), 199
GetWindowByIndex(), 199
GetWindowByName(), 199
GetWindowList(), 199
GetWindows(), 200
GetWinList(), 200
GetWins(), 200
GetWorkspaceInfo(), 304
GetWorkspaceInformation(), 304
GetWorkspaces(), 305
GetWses(), 305
GetWsInfo(), 304
GHS_AbortExecFile class
 __init__(), 339
GHS_AbortExecOnSignal class
 __init__(), 339
GHS_Action class
 attributes, 291
 DumpTree(), 291
GHS_ActionSequence class
 attributes, 292
 DumpTree(), 292
 Search(), 292
GHS_CoBrowse class
 __init__(), 330
 OpenCheckoutBrowserWindow(), 331
GHS_Debugger class
 __init__(), 225
 RunCommands(), 226
GHS_DebuggerApi class
 attributes, 238
 BigEndianTarget(), 245
 ChangeBreakpointInheritance(), 226
 ChangeDebugChildren(), 227
 ChangeDebugOnTaskCreation(), 227
 ChangeHaltOnAttach(), 228
 ChangeInheritProcessBits(), 229
 ChangeRunOnDetach(), 229
 ChangeStopAfterExec(), 230
 ChangeStopAfterFork(), 230
 ChangeStopOnTaskCreation(), 231
 CheckBreakpointInheritance(), 232
 CheckDebugChildren(), 232
 CheckDebugOnTaskCreation(), 232
 CheckHaltOnAttach(), 233
 CheckInheritProcessBits(), 233
 CheckRunOnDetach(), 233
 CheckStopAfterExec(), 234
 CheckStopAfterFork(), 234
 CheckStopOnTaskCreation(), 234
 CheckSymbol(), 244
 ConnectToRtserve(), 213
 ConnectToRtserve2(), 214
 ConnectToTarget(), 214
 DebugProgram(), 239
 Disconnect(), 215
 GetCpuFamily(), 246
 GetCpuMinor(), 246

Index

GetHostOsName(), 235
GetMultiVersion(), 235
GetPc(), 240
GetProgram(), 240
GetStatus(), 240
GetSymbolAddress(), 245
GetSymbolSize(), 245
GetTargetCoProcessor(), 246
GetTargetCpuFamilyName(), 247
GetTargetId(), 247
GetTargetOsMinorType(), 247
GetTargetOsName(), 247
GetTargetOsType(), 248
GetTargetPid(), 240
GetTargetSeries(), 248
Halt(), 240
IsConnected(), 215
IsFreezeMode(), 248
IsHalted(), 241
IsNativeDebugging(), 248
IsRunMode(), 249
IsRunning(), 241
IsStarted(), 241
Kill(), 242
Next(), 242
ReadIndirectValue(), 236
ReadIntegerFromMemory(), 236
ReadStringFromMemory(), 237
Resume(), 243
ShowConnectionOrganizerWindow(), 216
ShowOsaExplorerWindow(), 249
ShowTaskManagerWindow(), 250
ShowTraceWindow(), 250
Step(), 243
WaitToStop(), 244
WriteIntegerToMemory(), 237
WriteStringToMemory(), 238
GHS_DebuggerWindow class
 DumpToFile(), 254
 PrintFile(), 254
 PrintWindow(), 255
 RemoveBreakpoint(), 251
 RunCommands(), 250
 SetBreakpoint(), 252
 SetGroupBreakpoint(), 252
 ShowBreakpoints(), 253
 ShowBreakpointWindow(), 253
GHS_DebugServer class
 Disconnect(), 217
 GetComponent(), 217
 __init__(), 216
 LoadProgram(), 217
 RunCommands(), 218
 ShowTaskManagerWindow(), 218
GHS_DiffView class
 DiffFiles(), 332
 __init__(), 331
 OpenChooseWindow(), 332
GHS_DiffViewWindow class
 OpenDiff(), 333
 OpenNewDiff(), 333
 ShowCurrentDiff(), 333
 ToggleCaseSensitive(), 334
 ToggleCharDiff(), 334
 ToggleIgnoreAllWhiteSpace(), 334
 ToggleIgnoreCWhiteSpace(), 335
 ToggleIgnoreWhiteSpaceAmount(), 335
 ToggleLineupColumns(), 335
 ToggleNumber(), 336
 ToggleWordDiff(), 336
GHS_Editor class
 EditFile(), 267
 GotoLine(), 267
 __init__(), 266
GHS_EditorWindow class
 AddString(), 268
 Checkin(), 276
 Checkout(), 276
 CloseCurrentFile(), 271
 Copy(), 268
 Cut(), 269
 FlashCursor(), 273

Index

- GetSelectedString(), 274
- GetSelection(), 273
- GetTextLines(), 269
- GetTextString(), 269
- GotoNextFile(), 271
- GotoPrevFile(), 271
- MoveCursor(), 274
- OpenFile(), 272
- Paste(), 270
- PlaceUnderVC(), 276
- Redo(), 270
- SaveAsFile(), 272
- SaveIntoFile(), 273
- SelectAll(), 275
- SelectRange(), 275
- Undo(), 270
- GHS_EventAnalyzer class
 - CloseFile(), 279
 - GetFileList(), 279
 - __init__(), 278
 - OpenFile(), 279
 - ScrollToPosition(), 280
- GHS_EventAnalyzerWindow class
 - AutoTimeUnit(), 286
 - ChangeTimeUnit(), 286
 - CloseFile(), 281
 - GotoFirstView(), 282
 - GotoLastView(), 282
 - GotoNextView(), 283
 - GotoPrevView(), 283
 - NewWindow(), 287
 - OpenFile(), 281
 - SaveMevConfiguration(), 287
 - SelectRange(), 284
 - ShowLegend(), 287
 - ToggleFlatView(), 284
 - ViewRange(), 285
 - ZoomIn(), 285
 - ZoomOut(), 285
 - ZoomToSelection(), 285
- GHS_Exception class
 - __init__(), 340
- GHS_ExecFile() utility function, 92
- GHS_IdeObject class
 - attributes, 32
 - CleanCmdExecVariables(), 98
 - IsAlive(), 98
- GHS_Launcher class
 - __init__(), 297
- GHS_LauncherWindow class
 - AddAction(), 301
 - AddVariable(), 307
 - ChangeVariable(), 308
 - CreateWorkspace(), 303
 - DeleteAction(), 302
 - DeleteVariable(), 308
 - DeleteWorkspace(), 303
 - GetRunningActions(), 298
 - GetWorkspaceInformation(), 304
 - GetWorkspaces(), 305
 - LoadWorkspaceFile(), 305
 - RunAction(), 298
 - RunWorkspaceAction(), 299
 - SaveWorkspaceIntoFile(), 306
 - SelectWorkspace(), 306
 - WaitForActionsToFinish(), 300
- GHS_MemorySpaces class
 - attributes, 255
 - __init__(), 256
- GHS_MslTree class
 - attributes, 141
 - DumpTree(), 142
 - GetChildrenNumber(), 142
 - IsExpandable(), 143
 - IsExpanded(), 143
 - IsTopTree(), 143
 - SearchByColumnValue(), 143
 - SearchByName(), 144
 - SearchChildByColumnValue(), 145
 - SearchChildByName(), 145
 - SearchRow(), 146
- GHS_OsTypes class

Index

- attributes, 256
- __init__(), 257
- GHS_PrintObj() utility function, 93
- GHS_PrintObject() utility function, 93
- GHS_ProjectManager class
 - GetTopProjectFiles(), 313
 - __init__(), 312
 - OpenProject(), 313
- GHS_ProjectManagerWindow class
 - BuildAllProjects(), 314
 - BuildFile(), 314
 - BuildProjects(), 315
 - BuildSelectedProjects(), 316
 - CloseProject(), 318
 - ContractAll(), 323
 - ContractSelected(), 323
 - CopySelected(), 317
 - CutSelected(), 318
 - DebugSelectedProjects(), 322
 - DeleteSelected(), 318
 - DoubleClickTreeRow(), 325
 - EditSelectedProjects(), 322
 - ExpandAll(), 324
 - ExpandSelected(), 324
 - FindFile(), 321
 - GetStatus(), 316
 - HaltBuild(), 316
 - NewWindow(), 319
 - NextFile(), 321
 - OpenProject(), 319
 - PasteAfterSelected(), 318
 - PrintAll(), 324
 - PrintView(), 325
 - RevertFromFile(), 320
 - SaveChanges(), 320
 - SelectAll(), 326
 - SelectProject(), 326
 - SelectTreeRow(), 327
 - WaitForBuildingFinish(), 317
- GHS_RunShellCmds() utility function, 94
- GHS_RunShellCommands() utility function, 93
- GHS_Shell() utility function, 94
- GHS_ShellCmds() utility function, 94
- GHS_System() utility function, 94
- GHS_TargetIds class
 - __init__(), 257
- GHS_Task class
 - Attach(), 260
 - Detach(), 260
 - Halt(), 261
 - __init__(), 258
 - Next(), 261
 - Resume(), 262
 - RunCommands(), 258
 - RunCommandsViaDebugServer(), 259
 - Step(), 262
- GHS_Terminal class
 - __init__(), 219
 - MakeConnection(), 219
- GHS_TerminalWindow class
 - ChangeBaudRate(), 220
 - Connect(), 220
 - Disconnect(), 220
 - SendBreak(), 221
- GHS_TraceWindow class
 - FlushTraceBuffer(), 263
 - JumpToTrigger(), 263
 - StartTracing(), 263
 - StopTracing(), 264
- GHS_Variable class
 - attributes, 293
 - DumpTree(), 293
- GHS_Window class
 - attributes, 34
 - Beep(), 108
 - ChangeMslTree(), 156
 - ChangePullDownValue(), 170
 - ChangeTextFieldValue(), 183
 - ChangeWholeMslTree(), 157
 - ChooseDir(), 108

Index

ChooseFile(), 109
ChooseFromList(), 110
ChooseWindowFromGui(), 110
ChooseYesNo(), 111
ClearDefaultConfigFile(), 103
CloseWindow(), 133
DoubleClickMslCell(), 157
DoubleClickMslCellByValue(), 158
DumpAll(), 146
DumpButton(), 151
DumpMenu(), 113
DumpMenuBar(), 114
DumpMslHighlight(), 158
DumpMslSelection(), 159
DumpMslValue(), 159
DumpPullDownMenu(), 170
DumpPullDownValue(), 171
DumpTabContents(), 175
DumpTabSelection(), 175
DumpTabValue(), 176
DumpTextCellValue(), 181
DumpTextFieldValue(), 183
DumpWidget(), 147
ExtendMslSelection(), 160
GetCommandToChangeMslTree(), 161
GetCommandToChangePullDownValue(), 171
GetCommandToChangeTextFieldValue(), 184
GetCommandToClickButton(), 152
GetCommandToDoubleClickMslCell(), 161
GetCommandToDoubleClickMslCell...(), 162
GetCommandToDumpButton(), 152
GetCommandToDumpMenu(), 114
GetCommandToDumpMenuBar(), 115
GetCommandToDumpMsl(), 163
GetCommandToDumpMslHighlight(), 163
GetCommandToDumpMslSelection(), 164
GetCommandToDumpPullDownMenu(), 172
GetCommandToDumpPullDownValue(), 172
GetCommandToDumpTab(), 176
GetCommandToDumpTabSelection(), 177
GetCommandToDumpTabValue(), 177
GetCommandToDumpText(), 180
GetCommandToDumpTextField(), 185
GetCommandToDumpWidget(), 147
GetCommandToDumpWindow(), 148
GetCommandToExtendMslSelection(), 164
GetCommandToRegisterModalDialog...(), 123
GetCommandToReturnOnTextField(), 185
GetCommandToSelectMenu(), 115
GetCommandToSelectMenuPath(), 116
GetCommandToSelectMslCell(), 165
GetCommandToSelectMslCellByValue(), 166
GetCommandToSelectPullDownMenu(), 173
GetCommandToSelectSubMenu(), 116
GetCommandToSelectSubSubMenu(), 117
GetCommandToSelectTab(), 178
GetCommandToShowWidgets(), 149
GetCommandToSortMsl(), 166
GetCwd(), 101
GetDimension(), 133
GetEditTextLines(), 155
GetEditTextString(), 155
GetInfo(), 101
GetInput(), 111
GetIntegrityDistributionDir(), 105
GetLatestDir(), 106
GetMslRowNumber(), 167
GetMslTree(), 167
GetName(), 133
GetPid(), 101
GetPosition(), 134
GetPullDownMenu(), 173
GetPullDownValue(), 174
GetTabNames(), 179
GetTabSelection(), 179
GetTextCellValue(), 182
GetTextFieldValue(), 186
GetTextValue(), 181
GetUvelocityDistributionDir(), 106
IconifyWindow(), 134
IsButtonDimmed(), 153
IsButtonDown(), 153

Index

- IsIconified(), 135
- IsMenuItemActive(), 118
- IsMenuItemTicked(), 118
- IsSameWindow(), 102
- IsSubMenuItemActive(), 119
- IsSubMenuItemTicked(), 119
- IsTextCellReadOnly(), 182
- IsTextFieldReadOnly(), 186
- IsWindowAlive(), 102
- LoadConfigFile(), 103
- MoveWindow(), 135
- RecordGuiOperations(), 132
- RegisterModalDialogCommands(), 124
- RegisterModalDialogToChange...(), 124
- RegisterModalDialogToClick...(), 125
- RegisterModalDialogToDouble...(), 125
- RegisterModalDialogToDump...(), 126
 - to 127
- RegisterModalDialogToSelectM...(), 127
 - to 128
- RegisterModalDialogToSelectPull...(), 129
- RegisterModalDialogToShow...(), 129
- RegisterModalDialogToSortMsl(), 130
- RemoveRegisteredModalDialogCom...(), 130
- RenameWindow(), 135
- ResizeWindow(), 136
- RestoreWindow(), 137
- ReturnOnTextField(), 186
- RunCommands(), 102
- SaveConfig(), 104
- SelectButton(), 153
- SelectMenu(), 120
- SelectMslCell(), 168
- SelectMslCellByValue(), 168
- SelectPullDownValue(), 174
- SelectSubMenu(), 120
- SelectSubSubMenu(), 121
- SelectTab(), 179
- SetIntegrityDistributionDir(), 105
- SetLatestDir(), 107
- SetUvelocityDistributionDir(), 106
- ShowAttributes(), 137
- ShowConfigWindow(), 105
- ShowMessage(), 112
- ShowRegisteredModalDialogCom...(), 131
- ShowWidgets(), 149
- SortMslByColumn(), 169
- Wait(), 112
- WaitButtonInStatus(), 154
- WaitForMenuItem(), 122
- GHS_WindowClassNames class
 - attributes, 340
 - __init__(), 341
- GHS_WindowRegister class
 - Beep(), 201
 - CheckWindow(), 191
 - CheckWindowObject(), 192
 - ChooseDir(), 201
 - ChooseFile(), 202
 - ChooseFromList(), 203
 - ChooseWindowFromGui(), 203
 - ChooseYesNo(), 204
 - CloseAllWindows(), 206
 - GetCheckoutBrowserWindow(), 193
 - GetConnectionOrganizerWindow(), 194
 - GetDebuggerWindow(), 194
 - GetDialogByName(), 194
 - GetDiffViewerWindow(), 195
 - GetEditorWindow(), 195
 - GetEventAnalyzerWindow(), 195
 - GetHelpViewerWindow(), 196
 - GetInput(), 205
 - GetLauncherWindow(), 196
 - GetOsaExplorerWindow(), 196
 - GetProjectManagerWindow(), 197
 - GetPythonGuiWindow(), 197
 - GetResourceAnalyzerWindow(), 197
 - GetTaskManagerWindow(), 198
 - GetTerminalWindow(), 198
 - GetTraceWindow(), 198
 - GetWindowByIndex(), 199
 - GetWindowByName(), 199

Index

GetWindowList(), 199
IconifyAllWindows(), 206
 __init__(), 190
IsWindowInList(), 192
RestoreAllWindows(), 206
ShowMessage(), 205
ShowWindowList(), 200
WaitForWindow(), 207
WaitForWindowFromClass(), 208
WaitForWindowGoAway(), 208
WaitForWindowObjectGoAway(), 209
GHS_Workspace class
 attributes, 294
 DumpTree(), 295
 SearchAction(), 295
 SearchActionSequence(), 296
 SearchVariable(), 296
\$gi Py pane command, 50
-global command line option, 53
Goto(), 268
GotoFirstView(), 282
GotoLastView(), 282
GotoLine(), 267
GotoNextFile(), 271
GotoNextView(), 283
GotoPrevFile(), 271
GotoPrevView(), 283
graphical interfaces
 creating, 58, 85
GUI, *see* interfaces

H

-h command line option, 53
\$h Py pane command, 51
Halt()
 GHS_DebuggerApi class, 240
 GHS_ProjectManagerWindow
 class, 317
 GHS_Task class, 261
HaltBuild(), 316

HasChild(), 241
help, *see* online help
-help command line option, 53
Help menu
 online help, obtaining, 6
\$help Py pane command, 51
Help Viewer
 navigating, 8 to 10
 opening additional manuals in, 11
 window, 7

helpview command, 6
hierarchy of classes, 37
HostOsName(), 235

I

\$i Py pane command, 51
IconifyAllWindows(), 206
IconifyWin(), 134
IconifyWindow(), 134
IconifyWindows(), 206
IconifyWins(), 206
IconWin(), 134
IconWindow(), 134
IconWins(), 206
IgnoreAllSpace(), 335
IgnoreCSpace(), 335
IgnoreSpaceAmount(), 335
__init__()
 GHS_AbortExecFile class, 339
 GHS_AbortExecOnSignal class, 339
 GHS_CoBrowse class, 330
 GHS_Debugger class, 225
 GHS_DebugServer class, 216
 GHS_DiffView class, 331
 GHS_Editor class, 266
 GHS_EventAnalyzer class, 278
 GHS_Exception class, 340
 GHS_Launcher class, 297
 GHS_MemorySpaces class, 256
 GHS_OsTypes class, 257

Index

- GHS_ProjectManager class, 312
- GHS_TargetIds class, 257
- GHS_Task class, 258
- GHS_Terminal class, 219
- GHS_WindowClassNames class, 341
- GHS_WindowRegister class, 190
- input() Python function, 59
- InRunMode(), 249
- installing
 - additional Python modules, 31
 - Python, 30
- Integrated Development Environment (IDE), *see* MULTI Integrated Development Environment (IDE)
- integration
 - MULTI-Python, 30
- \$interactive** Py pane command, 51
- interfaces
 - creating, 58, 85
 - MULTI-Python, 47
- IsAlive(), 98
- IsBtnDimmed(), 153
- IsBtnDown(), 153
- IsButtonDimmed(), 153
- IsButtonDown(), 153
- IsConnected(), 215
- IsExpandable(), 143
- IsExpanded(), 143
- IsFreezeMode(), 248
- IsHalted(), 241
- IsIconified(), 135
- IsMenuEntryActive(), 118
- IsMenuEntryTicked(), 119
- IsMenuItemActive(), 118
- IsMenuItemTicked(), 118
- IsMin(), 135
- IsMinimized(), 135
- IsNativeDebugging(), 248
- IsRunMode(), 249
- IsRunning(), 241
- IsSameWin(), 102

- IsSameWindow(), 102
- IsStarted(), 241
- IsStopMode(), 248
- IsStopped(), 241
- IsSubMenuEntryActive(), 119
- IsSubMenuEntryTicked(), 120
- IsSubMenuItemActive(), 119
- IsSubMenuItemTicked(), 119
- IsTcReadOnly(), 182
- IsTextCellReadOnly(), 182
- IsTextFieldReadOnly(), 186
- IsTfReadOnly(), 186
- IsTop(), 143
- IsTopTree(), 143
- IsWindowAlive(), 102
- IsWindowInList(), 192

J

- JumpToTrigger(), 263

K

- keyboard shortcuts, Py pane, 50
 - abort execution, 52
 - execute statements, 52
 - indent, 52
 - print arguments, 52
 - save statements, 52
 - unindent, 52
- Kill(), 242

L

- LastView(), 283
- Launcher, *see* MULTI Launcher
- Launcher functions, 290
- Legend(), 287
- Lines(), 269
- LineupCol(), 335
- Load(), 218
- LoadConfigFile(), 103
- LoadFile()

Index

- GHS_EditorWindow class, 272
- GHS_EventAnalyzerWindow class, 282
- GHS_ProjectManager class, 313
- GHS_ProjectManagerWindow class, 319
- LoadModule(), 218
- LoadProg(), 218
- LoadProgram(), 217
- LoadWorkspace(), 305
- LoadWorkspaceFile(), 305
- LoadWs(), 305
- LoadWsFile(), 305
- local** command line option, 54

M

- macros
 - in MULTI scripts, 16 to 17
- MakeConnection(), 219
- .mbs** setup scripts, 16, 24
- MinimizeWindow(), 134
- MinimizeWindows(), 206
- MinWin(), 134
- MinWins(), 206
- miscellaneous classes, 37
- miscellaneous functions, 338
- MoveCursor(), 274
- MoveTo()
 - GHS_EditorWindow class, 274
 - GHS_EventAnalyzer class, 281
- MoveWin(), 135
- MoveWindow(), 135
- mpythonrun utility program
 - command line options, 53
 - overview, 53
- MULTI Integrated Development Environment (IDE)
 - document set, xxi
 - online help, 6
 - overview, 2
- MULTI Launcher
 - overview, 4
- MULTI scripts
 - creating, 16 to 17
 - examples, 18
 - for common tasks, 18
 - .mbs**, 16, 24
 - overview, 16
 - .rc**, 16
 - running, 24
 - statements in, 16
 - syntax checking, 23
- MULTI-Python classes, *see* classes, MULTI-Python
- MULTI-Python compatibility, 32
- MULTI-Python environment
 - extending, 46
- MULTI-Python functions, *see* functions, MULTI-Python
- MULTI-Python integration
 - overview, 30
 - troubleshooting, 59
- MULTI-Python interfaces, 47
- MULTI-Python tutorials, *see* tutorials, MULTI-Python
- MULTI-Python utility functions, *see* utility functions, MULTI-Python
- MULTI-Python variables, *see* variables, MULTI-Python
- MultiVersion(), 235

N

- NativeDebugging(), 248
- NativeProg(), 248
- NewWin()
 - GHS_EventAnalyzerWindow class, 287
 - GHS_ProjectManagerWindow class, 319
- NewWindow()

Index

- GHS_EventAnalyzerWindow
 class, 287
- GHS_ProjectManagerWindow
 class, 319
- Next()
 - GHS_DebuggerApi class, 242
 - GHS_ProjectManagerWindow
 class, 322
 - GHS_Task class, 261
- NextFile()
 - GHS_EditorWindow class, 271
 - GHS_ProjectManagerWindow
 class, 321
- NextView(), 283
- noconsole** command line option, 54
- NumDiff(), 336
- O**
- online help, 6
 - See also* Help Viewer
 - for commands, 6
 - for configuration options, 6
 - context-sensitive, 6
 - limitations, 10
 - online manuals, 6
 - overview, 6
- Open()
 - GHS_Editor class, 267
 - GHS_EditorWindow class, 272
 - GHS_EventAnalyzer class, 280
 - GHS_ProjectManager class, 313
 - GHS_ProjectManagerWindow
 class, 319
- OpenCheckoutBrowserWin(), 331
- OpenCheckoutBrowserWindow(), 331
- OpenChooseWin(), 332
- OpenChooseWindow(), 332
- OpenCoBrowser(), 331
- OpenCoBrowserWin(), 331
- OpenCoBrowserWindow(), 331

- OpenDiff(), 333
- OpenFile()
 - GHS_Editor class, 267
 - GHS_EditorWindow class, 272
 - GHS_EventAnalyzer class, 279
 - GHS_EventAnalyzerWindow
 class, 281
 - GHS_ProjectManager class, 313
 - GHS_ProjectManagerWindow
 class, 319
- OpenNewDiff(), 333
- OpenProj()
 - GHS_ProjectManager class, 313
 - GHS_ProjectManagerWindow
 class, 319
- OpenProject()
 - GHS_ProjectManager class, 313
 - GHS_ProjectManagerWindow
 class, 319
- OpenWin(), 331
- OpenWindow(), 331
- OsMinor(), 247
- OsName(), 248
- OsType(), 248

P

- \$p** socket server command, 57
- Paste()
 - GHS_EditorWindow class, 270
 - GHS_ProjectManagerWindow
 class, 318
- PasteAfterSelected(), 318
- Pc(), 240
- Pid(), 101
- PlaceIntoVC(), 276
- PlaceUnderVC(), 276
- PressButton(), 154
- PrevFile(), 271
- PrevView(), 283
- Print()

Index

- GHS_DebuggerWindow class, 254
- GHS_ProjectManagerWindow class, 325
- PrintAll(), 324
- PrintFile(), 254
- PrintView(), 325
- PrintWin()
 - GHS_DebuggerWindow class, 255
- PrintWindow()
 - GHS_DebuggerWindow class, 255
- ProgName(), 240
- Project Manager functions, 312
- prompt** command line option, 54
- \$prompt** socket server command, 57
- Py pane commands and keyboard shortcuts, 50
 - See also* keyboard shortcuts, Py pane
- Python
 - additional modules, installing, 31
 - installing, 30
- Python input functions, 59
- Python scripts
 - running, 55
- Python statements
 - running, 55
- PyUnicodeUCS2_FromUnicode
 - undefined symbol, 59
- PyUnicodeUCS4_FromUnicode
 - undefined symbol, 59

Q

- \$q** socket server command, 57
- \$quit** socket server command, 57

R

- \$r** Py pane command, 51
- RaiseWin(), 137
- RaiseWindow(), 137
- raw_input() Python function, 59
- .rc** setup scripts, 16

- ReadIndInt(), 236
- ReadIndirectInt(), 236
- ReadIndirectInteger(), 236
- ReadIndirectValue(), 236
- reading from/writing to memory
 - example MULTI script, 21
- ReadInt(), 237
- ReadIntegerFromMemory(), 236
- ReadStr(), 237
- ReadStringFromMemory(), 237
- RecGuiOp(), 132
- RecGuiOps(), 132
- RecordGuiOperations(), 132
- Redo(), 270
- RegChangePdVal(), 125
- RegChangePdValue(), 125
- RegClickBut(), 125
- RegClickButton(), 125
- RegDblClickMslCell(), 126
- RegDlgCmd(), 124
- RegDlgCmds(), 124
- RegDumpWidget(), 127
- RegDumpWin(), 127
- registering Python installation, 31
- RegisterModalDialogCommands(), 124
- RegisterModalDialogToChangePull...(), 124
- RegisterModalDialogToClickButton(), 125
- RegisterModalDialogToDoubleClick...(), 125
- RegisterModalDialogToDumpWidget(), 126
- RegisterModalDialogToDumpWindow(), 127
- RegisterModalDialogToSelectMslCell(), 127
- RegisterModalDialogToSelectMslCell...(), 128
- RegisterModalDialogToSelectPull...(), 129
- RegisterModalDialogToShowWidgets(), 129
- RegisterModalDialogToSortMsl(), 130
- regression testing
 - example MULTI script, 19
- RegSelMslCell(), 128
- RegSelMslCellByVal(), 128
- RegSelPdMenu(), 129
- RegShowWidgets(), 130

Index

- RegSortMsl(), 130
- RemoveBp(), 251
- RemoveBreakpoint(), 251
- RemoveRegisteredModalDialogCom...(), 130
- RenameWin(), 136
- RenameWindow(), 135
- reserved variable names, 44
- ResizeWin(), 136
- ResizeWindow(), 136
- \$restart** Py pane command, 51
- RestoreAllWindows(), 206
- RestoreWin(), 137
- RestoreWindow(), 137
- RestoreWindows(), 206
- RestoreWins(), 206
- Resume()
 - GHS_DebuggerApi class, 243
 - GHS_Task class, 262
- RetOnTf(), 187
- ReturnOnTextField(), 186
- ReturnOnTf(), 187
- Revert(), 320
- RevertFromFile(), 320
- RmBp(), 251
- RmDlgCmd(), 131
- RmDlgCmds(), 131
- Rtserv(), 213
- Rtserv2(), 214
- Run()
 - GHS_DebuggerApi class, 243
 - GHS_Task class, 262
- RunAction(), 298
- RunCmd()
 - GHS_Debugger class, 226
 - GHS_DebuggerWindow class, 251
 - GHS_DebugServer class, 218
 - GHS_Task class, 259
 - GHS_Window class, 103
- RunCmds()
 - GHS_Debugger class, 226
 - GHS_DebuggerWindow class, 251
- GHS_DebugServer class, 218
- GHS_Task class, 259
- GHS_Window class, 103
- RunCmdViaDbserve(), 259
- RunCmdViaDebugServer(), 259
- RunCommand()
 - GHS_Debugger class, 226
 - GHS_DebuggerWindow class, 251
 - GHS_DebugServer class, 218
 - GHS_Task class, 259
 - GHS_Window class, 103
- RunCommands()
 - GHS_Debugger class, 226
 - GHS_DebuggerWindow class, 250
 - GHS_DebugServer class, 218
 - GHS_Task class, 258
 - GHS_Window class, 102
- RunCommandsViaDebugServer(), 259
- RunMode(), 249
- running
 - MULTI scripts, 24
 - Python statements and scripts, 55
- RunWorkspaceAction(), 299
- RunWsAction(), 300
- S**
- s command line option, 54
- \$s** Py pane command, 51
- SameWin(), 102
- \$save** Py pane command, 51
- Save()
 - GHS_EditorWindow class, 273
 - GHS_ProjectManagerWindow class, 320
- SaveAs()
 - GHS_EditorWindow class, 272
- SaveAsFile()
 - GHS_EditorWindow class, 272
- SaveChange(), 320
- SaveChanges(), 320

Index

- SaveConfig(), 104
- SaveFile()
 - GHS_EditorWindow class, 273
- SaveIntoFile(), 273
- SaveMevConfig(), 287
- SaveMevConfiguration(), 287
- SaveWorkspaceIntoFile(), 306
- SaveWs(), 306
- SaveWsIntoFile(), 306
- sc command, 23
- script command line option, 54
- scripts, *see* MULTI scripts, *see* Python scripts
- ScrollTo(), 281
- ScrollToPosition(), 280
- Search(), 292
- SearchAction(), 295
- SearchActionSeq(), 296
- SearchActionSequence(), 296
- SearchByColumnValue(), 143
- SearchByColVal(), 144
- SearchByName(), 144
- SearchChildByColumnValue(), 145
- SearchChildByColVal(), 145
- SearchChildByName(), 145
- SearchRow(), 146
- SearchVar(), 297
- SearchVariable(), 296
- Sel()
 - GHS_EditorWindow class, 275
 - GHS_EventAnalyzerWindow class, 284
- SelAll()
 - GHS_EditorWindow class, 275
 - GHS_ProjectManagerWindow class, 326
- Select()
 - GHS_EditorWindow class, 275
 - GHS_EventAnalyzerWindow class, 284
- SelectAll()
 - GHS_EditorWindow class, 275
 - GHS_ProjectManagerWindow class, 326
- SelectButton(), 153
- SelectMenu(), 120
- SelectMslCell(), 168
- SelectMslCellByValue(), 168
- SelectProject(), 326
- SelectPullDownValue(), 174
- SelectRange()
 - GHS_EditorWindow class, 275
 - GHS_EventAnalyzerWindow class, 284
- SelectSubMenu(), 120
- SelectSubSubMenu(), 121
- SelectTab(), 179
- SelectTreeRow(), 327
- SelectWorkspace(), 306
- SelMenu(), 120
- SelMslCell(), 168
- SelMslCellByValue(), 169
- SelPdVal(), 174
- SelPdValue(), 174
- SelProj(), 326
- SelRange()
 - GHS_EditorWindow class, 275
 - GHS_EventAnalyzerWindow class, 284
- SelSubMenu(), 121
- SelSubSubMenu(), 122
- SelTab(), 180
- SelTreeRow(), 327
- SelWs(), 307
- SendBreak(), 221
- Series(), 248
- service classes, 33
- SetBp(), 252
- SetBreakpoint(), 252
- SetGroupBreakpoint(), 252
- SetGrpBp(), 252
- SetIntDir(), 106

Index

- SetIntDistDir(), 106
- SetIntegrityDistributionDir(), 105
- SetLatestDir(), 107
- SetMruDir(), 107
- SetUvelDir(), 106
- SetUvelDistDir(), 106
- SetUvelocityDistributionDir(), 106
- ShowAttr(), 137
- ShowAttributes(), 137
- ShowBps(), 253
- ShowBpWin(), 253
- ShowBreakpoints(), 253
- ShowBreakpointWindow(), 253
- ShowCo(), 216
- ShowConfigWin(), 105
- ShowConfigWindow(), 105
- ShowConnectionOrganizer(), 216
- ShowConnectionOrganizerWindow(), 216
- ShowCurDiff(), 333
- ShowCurrentDiff(), 333
- ShowDlgCmd(), 131
- ShowDlgCmds(), 131
- ShowLegend(), 287
- ShowMessage()
 - GHS_Window class, 112
 - GHS_WindowRegister class, 205
- ShowMsg()
 - GHS_Window class, 112
 - GHS_WindowRegister class, 206
- ShowOsa(), 249
- ShowOsaExplorer(), 249
- ShowOsaExplorerWindow(), 249
- ShowRegisteredModalDialogCom...(), 131
- ShowTaskManagerWindow()
 - GHS_DebuggerApi class, 250
 - GHS_DebugServer class, 218
- ShowTaskWin()
 - GHS_DebuggerApi class, 250
 - GHS_DebugServer class, 218
- ShowTaskWindow()
 - GHS_DebuggerApi class, 250
 - GHS_DebugServer class, 218
- ShowTraceWin(), 250
- ShowTraceWindow(), 250
- ShowWidgets(), 149
- ShowWindowList(), 200
- ShowWindows(), 201
- ShowWins(), 201
- socket** command line option, 54
- socket servers
 - commands, 57
 - starting, 56
- SortMsl(), 169
- SortMslByCol(), 169
- SortMslByColumn(), 169
- starting
 - socket servers, 56
- StartTrace(), 264
- StartTracing(), 263
- statement** command line option, 54
- statements, *see* Python statements
- stdin**, 59
- Step()
 - GHS_DebuggerApi class, 243
 - GHS_Task class, 262
- Stop()
 - GHS_DebuggerApi class, 241
 - GHS_Task class, 261
- StopMode(), 248
- StopTrace(), 264
- StopTracing(), 264
- strings in scripts
 - example MULTI script, 22
- syntax checking
 - MULTI scripts, 23
- T**
- Target(), 215
- TargetPid(), 240
- TaskWin()
 - GHS_DebuggerApi class, 250

Index

- GHS_DebugServer class, 218
- TaskWindow()
 - GHS_DebuggerApi class, 250
 - GHS_DebugServer class, 218
- Tcl/Tk GUI package, 58, 85
- telnet** command line option, 54
- TermLines(), 155
- TermTextLines(), 155
- TermTextStr(), 156
- TermTextString(), 156
- TextLines(), 269
- TextStr(), 269
- TextString(), 269
- ToggleCaseSensitive(), 334
- ToggleCharDiff(), 334
- ToggleFlatView(), 284
- ToggleIgnoreAllWhiteSpace(), 334
- ToggleIgnoreCWhiteSpace(), 335
- ToggleIgnoreWhiteSpaceAmount(), 335
- ToggleLineupColumns(), 335
- ToggleNum(), 336
- ToggleNumber(), 336
- ToggleWordDiff(), 336
- TopProjs(), 313
- toupper()
 - example MULTI script, 22
- TraceOff(), 264
- TraceOn(), 264
- troubleshooting, 59
- tutorials, MULTI-Python, 62
 - manipulating Debugger, 77
 - manipulating Editor, 75
 - manipulating windows, 62
- typographical conventions, xxii

U

- undefined symbols, 59
- Undo(), 270
- Unit(), 286
- updating the source pane during execution

- example MULTI script, 21
- utility classes, 36
- utility functions, MULTI-Python, 43
 - GHS_ExecFile(), 92
 - GHS_PrintObject(), 93
 - GHS_RunShellCommands(), 93
 - GHS_System(), 94

V

- \$v** Py pane command, 51
- variables, MULTI-Python, 44
 - reserved names, 44
- verbose** command line option, 54
- \$verbose** Py pane command, 51
- version control functions, 330
- View(), 285
- ViewRange(), 285

W

- Wait()
 - GHS_LauncherWindow class, 300
 - GHS_Window class, 112
- WaitButtonInStatus(), 154
- WaitForActionsToFinish(), 300
- WaitForBuildingFinish(), 317
- WaitForMenuEntry(), 122
- WaitForMenuItem(), 122
- WaitForWindow(), 207
- WaitForWindowFromClass(), 208
- WaitForWindowGoAway(), 208
- WaitForWindowObjectGoAway(), 209
- WaitForWinGoAway(), 209
- WaitForWinObjGoAway(), 210
- WaitMenuEntry(), 122
- WaitMenuItem(), 122
- WaitToFinish(), 300
- WaitToStop(), 244
- WaitWin(), 208
- WaitWindow(), 208
- WaitWinFromClass(), 208

Index

- widget functions, 140
- Widgets(), 151
- window classes, 35
- window functions, 100
- window tracking functions, 190
- WindowForIndex(), 199
- Windows registry
 - registering Python installation, 31
- Windows(), 200
- WinFIdx(), 199
- WinForIdx(), 199
- Wins(), 200
- WordDiff(), 336
- workspaces
 - overview, 5
- WriteInt(), 238
- WriteIntegerToMemory(), 237
- WriteStr(), 238
- WriteString(), 238

- WriteStringToMemory(), 238
- writing to memory
 - example MULTI script, 21

Y

- YesNo()
 - GHS_Window class, 111
 - GHS_WindowRegister class, 204
- YesOrNo()
 - GHS_Window class, 111
 - GHS_WindowRegister class, 204

Z

- ZoomIn(), 285
- ZoomOut(), 285
- ZoomSel(), 285
- ZoomToSel(), 285
- ZoomToSelection(), 285